DS&SC-1

Initiation à la programmation (impérative) avec

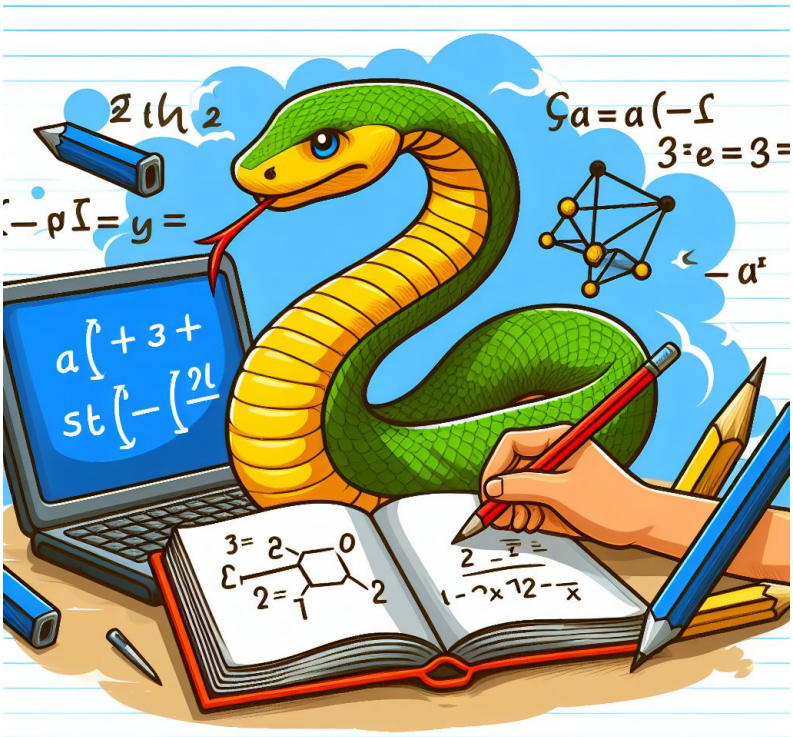
python

Recueil de 429 exercices corrigés et aide-mémoire.

Gloria Faccanoni

https://moodle.univ-tln.fr/course/view.php?id=4968

Année 2026 – 2027
Dernière mise-à-jour : Vendredi 23 octobre 2026



IMATH Bureau M-117
Université de Toulon
Avenue de l'université
83957 LA GARDE - FRANCE

☎ 0033 (0)4 83 16 66 72

✉ gloria.faccanoni@univ-tln.fr
http://faccanoni.univ-tln.fr

UNIVERSITÉ DE TOULON

Ces exercices ont pour but de pratiquer et de vérifier votre compréhension des notions du chapitre. L'important n'est pas de trouver la solution parfaite, mais de savoir mobiliser les concepts vus en cours.

Attention : **interdiction d'utiliser des notions non encore traitées.** Par exemple, les boucles seront vues au chapitre 4. Ainsi, les exercices des chapitres 1 à 3 doivent être résolus sans boucle et ont été conçus pour être résolus avec seulement les notions déjà étudiées.

Les IA (comme ChatGPT) peuvent aider à comprendre et expliquer les exercices, mais **ne doivent pas remplacer votre travail**. Tout d'abord, elles peuvent proposer des réponses fausses (rarement). Mais, même correctes, leurs réponses sont souvent trop complexes et hors programme. Comme le but est de vérifier si les notions ont été bien acquises, cette approche directe est inutile pour l'apprentissage.

Utilisation utile d'une IA : 1. faire l'exercice soi-même d'abord ; 2. comparer avec la solution proposée dans ce polycopié ; 3. si nécessaire, demander à une IA des explications sur la solution du polycopié.

Les exercices du programme indicatif de la page suivante suivent l'ordre de ce polycopié, chaque séance reprenant là où la précédente s'est arrêtée ; seule la dernière séance revient volontairement sur des chapitres antérieurs, pour réviser. **Les derniers exercices d'une séance peuvent ne pas être terminés** : ils sont là pour ceux qui avancent vite, et restent à faire chez soi sinon.

Tous les exercices sont à préparer à la maison avant la séance. La séance sert à les saisir, les tester, les corriger et poser des questions, non à les découvrir.

Programme Indicatif — L1 MATHS

Sem.	CM	TD	TP
37	Moodle Ch. 1		
38	Ch. 2-3		
39	Ch. 4-5-6		
40		1.1 - 1.2 - 1.3 - 2.2 - 2.8 - 2.11 - 2.12 - 2.22 - 2.36	
41	Ch. 7-8		
42		[3.1 - 3.2 -] 3.3 - 3.4 [4.1 - 4.18 -] 4.20	
43		5.1 - 5.2 - 5.4 - 5.10 - 5.34 - 5.39 6.1 - 6.2 - 6.11 - 6.39	
44		Pause	
45			1.5 - 1.7- 1.9 - 1.10 2.9 - 2.10 - 2.15 - 2.25 - 2.27 - 2.28 - 2.29 - 2.30 - 2.34 3.7 - 3.8 - 3.9 - 3.10
46		6.43 - 6.44 - 6.74 - 6.76 7.4 - 7.6 - 7.21 - 7.32 8.1 - 8.3 - 8.4 - 8.15 - 8.23	4.5 - 4.7 - 4.9 - 4.12 - 4.14 - 4.16 - 4.17 - 4.19 - 4.21 - 4.22 - 4.24 - 4.32 - 4.33 - 4.42 - 4.43 - 4.72 - 4.73 - 4.75 - 4.76
47			5.3 - 5.8 - 5.9 - 5.11 - 5.12 - 5.13 - 5.14 - 5.17 - 5.21 - 5.23 - 5.28 - 5.32 - 5.33 - 5.34 - 5.37 - 5.38 - 5.40
48			6.5 - 6.8 - 6.9 - 6.14 - 6.16 - 6.17 - 6.31 - 6.35 - 6.41 - 6.47 - 6.54 - 6.56 - 6.57 - 6.59 - 6.60 - 6.63 - 6.65 - 6.70 - 6.78 Test Moodle d'entraînement
49			7.9 - 7.10 - 7.11 - 7.13 - 7.15 - 7.20 - 7.22 - 7.28 - 7.30 - 7.36 - 7.37 - 7.44 - 7.45 - 7.49 Test Moodle d'entraînement
50			8.2 - 8.6 - 8.14 - 8.16 - 8.18 - 8.20 - 8.21 - 8.22 - 8.26
51			**Révisions**: 6.29 - 6.30 - 6.32 - 6.34 - 6.35 - 6.43 - 6.44 - 6.57 - 6.107 - 7.4 - 7.15 - 8.5 - 8.16
		CC — jeudi 17 décembre, en salle de TP	

CT (en salle de TP), début janvier (session 1) et fin juin (session 2 le cas échéant)

Programme Indicatif — MOCA 2A

Séance	Date	Exercices
1	Vendredi 23 octobre	1.10 - 1.11 2.8 - 2.11 - 2.14 - 2.15 - 2.22 - 2.25 -
2	Jeudi 5 novembre	3.4 - 3.10 4.5 - 4.9 - 4.16 - 4.17 - 4.19 - 4.21 - 4.24 - 4.29 - 4.32 - 4.33 - 4.42 - 4.43 - 4.56 - 4.58 - 4.72 - 4.73 - 4.75 - 4.76 5.1 - 5.3 - 5.8 - 5.9 - 5.10 - 5.12 - 5.13 - 5.14 - 5.17 - 5.21 - 5.23 - 5.28 - 5.33 - 5.34 - 5.37 - 5.38 - 5.40
3	Jeudi 12 novembre	6.5 - 6.11 - 6.26 - 6.30 - 6.33 - 6.34 - 6.35 - 6.47 - 6.17 - 6.54 - 6.57 - 6.59 - 6.60 - 6.63 - 6.67 - 6.70 - 6.76 - 6.83 - 6.84 - 6.122
4	Vendredi 20 novembre	7.3 - 7.6 - 7.8 - 7.10 - 7.11 - 7.12 - 7.15 - 7.20 - 7.22 - 7.25 - 7.28 - 7.37 - 7.43 - 7.44 - 7.45 8.1 - 8.2 - 8.5 - 8.7 - 8.10 - 8.14 - 8.16 - 8.18 - 8.20 - 8.21 - 8.26 - 8.27
5	Vendredi 26 novembre	Test Moodle d'entraînement

CC (en salle de TP), Lundi 30 novembre

Table des matières

Introduction **7**

1. Notions de base de Python **11**
1.1. Mode interactif et mode script . . . 11
1.2. Commentaires . . . 14
1.3. Indentation . . . 14
1.4. Types primitifs . . . 14
1.5. Variables et affectation . . . 15
1.6. Nombres et opérations arithmétiques . . . 18
1.7. Type booléen, Opérateurs de comparaison et connecteurs logiques . . . 19
1.8. Exercices . . . 21

2. Structures de référence **29**
2.1. Les chaînes de caractères (String) et la fonction `print` . . . 29
2.2. Listes . . . 39
2.3. Les tuples . . . 44
2.4. L'itérateur `range` . . . 45
2.5. ☆ Les dictionnaires (ou tableaux associatifs) . . . 46
2.6. ☆ Les ensembles . . . 48
2.7. Exercices . . . 51

3. Déroulement d'un programme : instructions conditionnelles **75**
3.1. Définir des conditions avec les instructions `if/elif/else` . . . 75
3.2. Exercices . . . 79

4. Structures itératives **87**
4.1. Boucle `for` : répétition déterminée (parcourir) . . . 87
4.2. Boucle `while` : répétition indéterminée (conditionnelle) . . . 89
4.3. Suites . . . 89
4.4. ☆ Ruptures de séquences . . . 92
4.5. Exercices . . . 95

5. Définitions en compréhension **137**
5.1. Listes en compréhension . . . 137
5.2. ☆ Dictionnaires en compréhension . . . 138
5.3. ☆ Ensembles en compréhension . . . 139
5.4. Exercices . . . 141

6. Fonctions **165**
6.1. Fonctions prédéfinies . . . 165
6.2. Définition et utilisation d'une fonction personnelle . . . 167
6.3. Fonctions Lambda (fonctions anonymes) . . . 171
6.4. ☆ Indication des types des variables à l'entrée et à la sortie des fonctions . . . 172
6.5. ☆ Arguments par défaut . . . 173
6.6. ☆ Fonctions récursives . . . 173
6.7. Exercices . . . 175

7. Modules **265**
7.1. Importation d'un module . . . 265
7.2. Quelques modules mathématiques courants . . . 267
7.3. Quelques autres modules courants . . . 272
7.4. ☆ Modules personnels . . . 276
7.5. Exercices . . . 277

8. Introduction à Matplotlib pour les tracés de base **325**
8.1. Importation des modules `matplotlib` et `numpy` . . . 326
8.2. Tracé d'une courbe sur un repère . . . 327
8.3. Plusieurs courbes sur le même repère et options . . . 328
8.4. Plusieurs repères dans des fenêtres distinctes . . . 330
8.5. Plusieurs repères dans une grille au sein d'une même fenêtre . . . 331
8.6. ☆ Animations . . . 332
8.7. ☆ Régression : polynôme de meilleure approximation . . . 332
8.8. Exercices . . . 335

A. Les «mauvaises» propriétés des nombres flottants et la notion de précision **367**
A.1. Quelques remarques sur les nombres en Virgule Flottante . . . 367
A.2. Ne jamais faire confiance aveuglément aux résultats d'un calcul obtenu avec un ordinateur... . . . 371
A.3. Exercices . . . 375

Introduction

Les **mathématiques appliquées** sont une discipline qui se situe à la croisée de nombreux domaines scientifiques, englobant la statistique, l'informatique, la physique, la chimie, la mécanique, la biologie, l'économie, les sciences de l'ingénieur, et bien d'autres encore. En appliquant les mathématiques à ces domaines variés, on se retrouve souvent confronté à des problèmes complexes qui ne peuvent être résolus de manière exacte par des méthodes analytiques traditionnelles, voire qui sont simplement difficiles à résoudre par des calculs manuels. Dans de telles situations, le recours aux calculs informatiques devient essentiel.

Un outil de programmation adapté à ce travail doit posséder plusieurs caractéristiques clés :

- une large collection d'algorithmes et d'outils de base disponibles ;
- une facilité d'apprentissage, idéalement proche du langage naturel des mathématiques ;
- un environnement ou langage unique pour toutes les problématiques (simulations, traitement des données, visualisation) ;
- une efficacité dans l'exécution et le développement ;
- une facilité de communication avec les collaborateurs, les étudiants, les clients, etc.

Python répond à ces critères : il offre une grande variété de bibliothèques scientifiques, ainsi que des outils adaptés à des tâches non scientifiques. C'est un langage de programmation bien conçu et lisible, disponible gratuitement en tant que logiciel open source.

Parcours Data Science and Scientific Computing

Depuis septembre 2024, l'université de Toulon propose un parcours innovant au sein de sa licence de Mathématiques : le parcours **Data Science and Scientific Computing**. Il se situe à l'intersection des mathématiques et de l'informatique, visant à former les étudiants à formuler des questions complexes dont les réponses, exactes ou approchées, sont obtenues par le calcul informatique.

Ce programme de trois ans offre une formation théorique et pratique en calcul scientifique et en analyse de données, en mettant l'accent sur l'utilisation de **Python** et des **Notebook Jupyter**. On y apprend à manipuler, analyser et visualiser les données, ainsi qu'à résoudre des problèmes scientifiques et mathématiques complexes en utilisant les outils et bibliothèques appropriés. Voici les 6 modules qui le composent :

DSSC-1 **Introduction à la programmation scientifique avec Python.** Ce cours couvre les bases de la programmation en Python : les types de données, les structures de contrôle, les fonctions (fonctions intégrées, modules, et fonctions personnalisées) ainsi que l'utilisation de Matplotlib pour créer des visualisations.
https://moodle.univ-tln.fr/course/view.php?id=4968

DSSC-2 **Initiation au calcul scientifique et au notebook Jupyter.** Ce cours introduit les notebooks, qui permettent de créer des documents interactifs combinant du code Python et du texte en Markdown. Le programme couvre la rédaction en Markdown avec des formules LaTeX, et l'utilisation de Python pour résoudre des problèmes mathématiques tels que la résolution d'équations non linéaires, l'approximation des zéros d'une fonction, l'interpolation pour estimer des valeurs manquantes, et l'approximation d'intégrales.
https://moodle.univ-tln.fr/enrol/index.php?id=4671

DSSC-3 **Introduction au calcul formel/symbolique avec SymPy et rédaction scientifique avec LaTeX.** Ce module commence par une révision du calcul différentiel et de l'algèbre linéaire en utilisant SymPy pour le calcul symbolique dans des notebooks Jupyter. Ensuite, étant donné que LaTeX est un langage incontournable pour la composition typographique dans les domaines scientifiques, le cours abordera également la création de documents complets en LaTeX (en effet, dans les notebooks, il n'est utilisé que pour formater les équations scientifiques).
https://moodle.univ-tln.fr/course/view.php?id=8894

DSSC-4 **Manipulation avancée des tableaux avec NumPy.** Ce cours introduit le calcul matriciel avec NumPy, un module puissant pour effectuer efficacement des opérations complexes sur des tableaux multidimensionnels.
https://moodle.univ-tln.fr/enrol/index.php?id=6844

DSSC-5 **Introduction à la manipulation de données avec Pandas.** Ce module enseigne l'utilisation de Pandas pour la manipulation efficace des jeux de données, y compris l'importation, le nettoyage, la transformation, le filtrage, l'agrégation, et la gestion des valeurs manquantes.
https://moodle.univ-tln.fr/course/view.php?id=8889

DSSC-6 **Approximations d'équations différentielles (problèmes de Cauchy).** Ce cours aborde les bases de l'approximation des équations différentielles. Dans un premier temps, les résultats approchés obtenus avec la bibliothèque SciPy seront comparés aux solutions exactes fournies par SymPy. Ensuite, le cours explorera de manière théorique et pratique plusieurs méthodes d'approximation, telles que les schémas classiques, les schémas de type Runge-Kutta, et les schémas multipas.
https://moodle.univ-tln.fr/course/view.php?id=4840

Déroulement du cours DSSC-1 et évaluation

Ce polycopié est associé au premier cours, initiant à la programmation scientifique avec Python. Il vise à enseigner deux compétences essentielles : d'une part, la résolution de problèmes, souvent issus des mathématiques, en utilisant des algorithmes, et d'autre part, la capacité à coder ces algorithmes en Python. En effet, le terme "programmation" englobe deux activités distinctes : l'analyse du problème pour élaborer un algorithme approprié, puis sa traduction dans un langage compréhensible par l'ordinateur, en l'occurrence Python.

Le cours se compose de :

CM 6h : 4 séances de 1h30
TD 6h : 4 séances de 1h30
TP 21h : 7 séances de 3h

CC 3h : 1 séance de 3h en salle de TP la dernière semaine de cours
CT 3h : 1 séance de 3h en salle de TP en janvier
Note finale $\max(\mathrm{CT}; 0.3\mathrm{CC} + 0.7\mathrm{CT})$

Les CC et CT prennent la forme de tests Moodle, avec **une série de questions tirées aléatoirement dans une même banque de questions**. La correction est **automatique**, demandant aux étudiants d'écrire des codes Python (scripts ou fonctions) qui seront vérifiés sur plusieurs tests tirés aléatoirement.

Comment utiliser le polycopié

À l'université, il est facile de sous-estimer l'importance de commencer à travailler dès le début du semestre. Toutefois, reporter votre travail peut entraîner une accumulation de retard difficile à rattraper, compromettant ainsi vos chances de succès. Pour éviter de vous retrouver dans une situation où il serait trop tard pour bien faire, il est essentiel de **comprendre** et **assimiler** le cours **au fur et à mesure**, tout en pratiquant en parallèle. N'hésitez pas à recourir à la célèbre méthode du canard en plastique.

Ce polycopié ne remplace pas les cours, TD et TP, mais vise à faciliter la prise de notes pendant les séances et à permettre aux étudiants de se concentrer sur les explications orales. Cependant, il ne contient pas toutes les informations présentées durant le cours et peut comporter des erreurs. Si vous repérez des erreurs, merci de les signaler.

En complément, ce document propose **429 exercices** de difficulté variée, dont les corrections sont disponibles sur la page Moodle [1]. Il est normal que votre code diffère de celui proposé dans la correction, car il existe souvent plusieurs façons de résoudre un même problème. L'important est de comprendre la démarche qui mène à l'analyse du problème et à la formulation d'un algorithme, plutôt que de rechercher une solution unique.

Conventions pour la présentation des exercices et du code

Les exercices marqués par le symbole devront être préparés avant les TP (certains auront été traités en TD). Ceux marqués par le symbole sont un peu plus difficiles et ne seront pas traités en TD (ni en TP sauf si tous les autres exercices sont terminés). Les exercices marqués par le symbole sont des Pydéfis. La correction de ces exercices n'est pas publique, mais je vous conseille de vous inscrire sur le site et de vérifier vos réponses.

Concernant la présentation du code, les instructions précédées de trois chevrons (>>>) sont à saisir dans une session interactive (si l'instruction produit un résultat, il est affiché une fois l'instruction exécutée) tandis que les instructions sans chevrons sont des bouts de code à écrire dans un fichier, avec le résultat d'exécution du script affiché juste après.

Version de référence

La version de Python utilisée pour ce polycopié est la version 3.12 . Veuillez noter que certaines constructions telles que les `f-strings` peuvent ne pas être disponibles dans des versions antérieures de Python.

Utiliser Python en ligne

Il existe plusieurs sites où l'on peut écrire et tester ses propres programmes, sans nécessité de téléchargement ou d'installation. Ces sites sont utiles pour des tests ou pour apprendre Python sans installer de logiciel sur votre machine. En particulier :

- pour écrire et exécuter des script contenant des graphes `matplotlib`
  https://console.basthon.fr/
- pour comprendre l'exécution d'un code pas à pas : Visualize code and get live help
  http://pythontutor.com/visualize.html

Obtenir et installer Python

Python peut être téléchargé depuis le site officiel www.python.org. Pour l'installer, suivez les instructions spécifiques à votre système d'exploitation (Windows ou Mac). Pour ce qui est des systèmes Linux, il est très probable que Python soit déjà installé.

1. https://moodle.univ-tln.fr/course/view.php?id=4968

Vous pouvez sinon installer Anaconda, incluant Python ainsi que des modules utiles en mathématiques (`Matplotlib`, `NumPy`, `SciPy`, `SymPy`, etc.). Les instructions d'installation spécifiques à chaque système d'exploitation sont disponibles sur https://www.anaconda.com/products/individual.

Quel éditeur ?

Pour écrire des scripts Python, utilisez **un éditeur de texte pur**, pas Word ou Libre Office Writer. Des éditeurs comme NOTEPAD, NOTEPAD++, EDIT, GEDIT, GEANY, KATE, VI, VIM, EMACS, NANO, SUBLIME TEXT peuvent convenir selon votre système d'exploitation.

Vous pouvez aussi utiliser **des environnements de développement spécialisés** (appelés **IDE** pour *Integrated Development Environment*) tels que IDLE, THONNY ou SPYDER. Ces derniers se présentent sous la forme d'une application et se composent généralement d'un éditeur de code, d'une fenêtre appelée indifféremment *console*, *shell* ou *terminal* Python, d'un débogueur et d'un générateur d'interface graphique.

L'IDE **THONNY**[2] installe Python 3.10 en même temps et c'est peut-être le choix plus simple pour commencer (notamment sous Windows). Il gère aussi l'installation de modules tels que `scipy`, `sympy` et `matplotlib`.

Si vous n'êtes plus des néophytes en programmation, vous trouverez votre bonheur avec VSCODE[3].

Le Zen de Python

La PEP 20 est une sorte de réflexion philosophique avec des phrases simples qui devraient guider tout programmeur. Comme les développeurs de Python ne manquent pas d'humour, celle-ci est accessible sous la forme d'un "œuf de Pâques" (*easter egg* en anglais) en important un module nommé `this`. On l'appelle le "Zen de Python" et résume la philosophie du langage :

The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

2. https://thonny.org/ et https://realpython.com/python-thonny/
3. https://docs.microsoft.com/fr-fr/learn/modules/python-install-vscode/1-introduction

CHAPITRE 1

Notions de base de Python

Python est un langage développé en 1 989 par Guido VAN ROSSUM, aux Pays-Bas. Le nom est dérivé de la série télévisée britannique des *Monty Python's Flying Circus*. La dernière version de Python est la version 3. Plus précisément, la version 3.12 a été publiée en décembre 2023. [1] La Python Software Foundation est l'association qui organise le développement de Python et anime la communauté de développeurs et d'utilisateurs.

Ce langage de programmation offre plusieurs caractéristiques notables :

- **Multi-plateforme** : Fonctionnant sur divers systèmes d'exploitation tels que Linux, Mac OS X, Windows, ou même Android, un programme écrit sur un système peut s'exécuter sans modification sur d'autres systèmes.
- **Gratuité** : Python peut être installé sur autant d'ordinateurs que nécessaire, y compris sur des appareils mobiles.
- **Accessibilité** : Python est un langage de haut niveau, demandant peu de connaissances sur le fonctionnement interne d'un ordinateur pour être utilisé.
- **Interprété** : Les programmes Python ne nécessitent pas de compilation en code machine et sont exécutés via un interpréteur, offrant une facilité de test et de débogage rapide.
- **Simplicité** : Relativement facile à apprendre et à utiliser.
- **Utilisation étendue** : Très populaire dans les domaines scientifiques et l'analyse de données.

Ces caractéristiques font de Python un langage largement enseigné, de l'éducation secondaire à l'enseignement supérieur.

IDE (pour *Integrated Development Environment*) : Idle et Thonny

Les *Integrated Development Environment* se présentent sous la forme d'une application et se composent d'une fenêtre appelée indifféremment *console*, *shell* ou *terminal* Python, et d'un éditeur de code. Dans les salles de TP nous utiliserons l'IDE IDLE. [2] Si vous voulez utiliser vos ordinateurs personnels, je conseille d'installer l'IDE THONNY. [3]

1.1. Mode interactif et mode script

L'exécution d'un programme Python repose sur un *interpréteur*, un logiciel qui traduit les instructions écrites en Python en langage machine, permettant ainsi leur exécution directe par l'ordinateur. Cette traduction est effectuée à la volée, semblable à la traduction en temps réel des discours des parlementaires lors des sessions du Parlement européen par les interprètes. Ainsi, Python est classifié comme un *langage interprété*.

Il existe *deux modes d'utilisation de Python* :

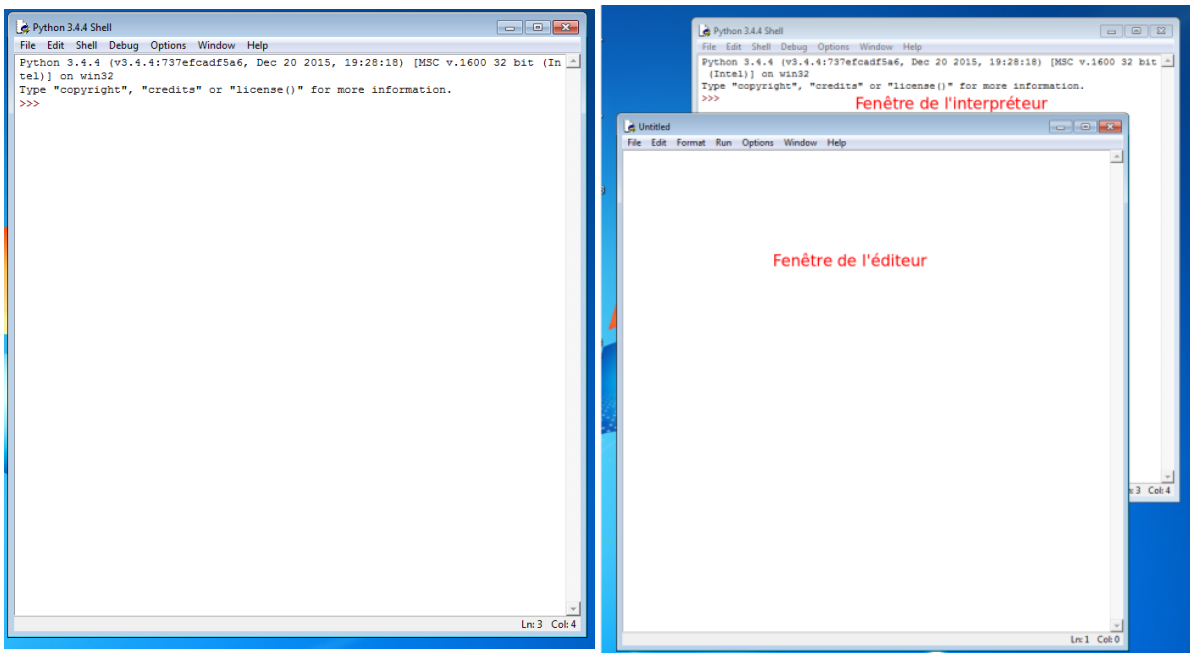
- Le **mode interactif**, également appelé **mode console**, **mode shell** ou **terminal Python**, permet d'encoder les instructions une par une. Après chaque saisie d'instruction, l'interpréteur exécute celle-ci lorsqu'on presse la touche «Entrée».
- Le **mode script** nécessite l'écriture préalable de toutes les instructions du programme dans un fichier texte portant l'extension `.py`. Par la suite, on demande à Python de lire ce fichier et exécuter son contenu, pas à pas, simulant ainsi une saisie successive des instructions comme dans le mode interactif.

1.1.1. Mode interactif

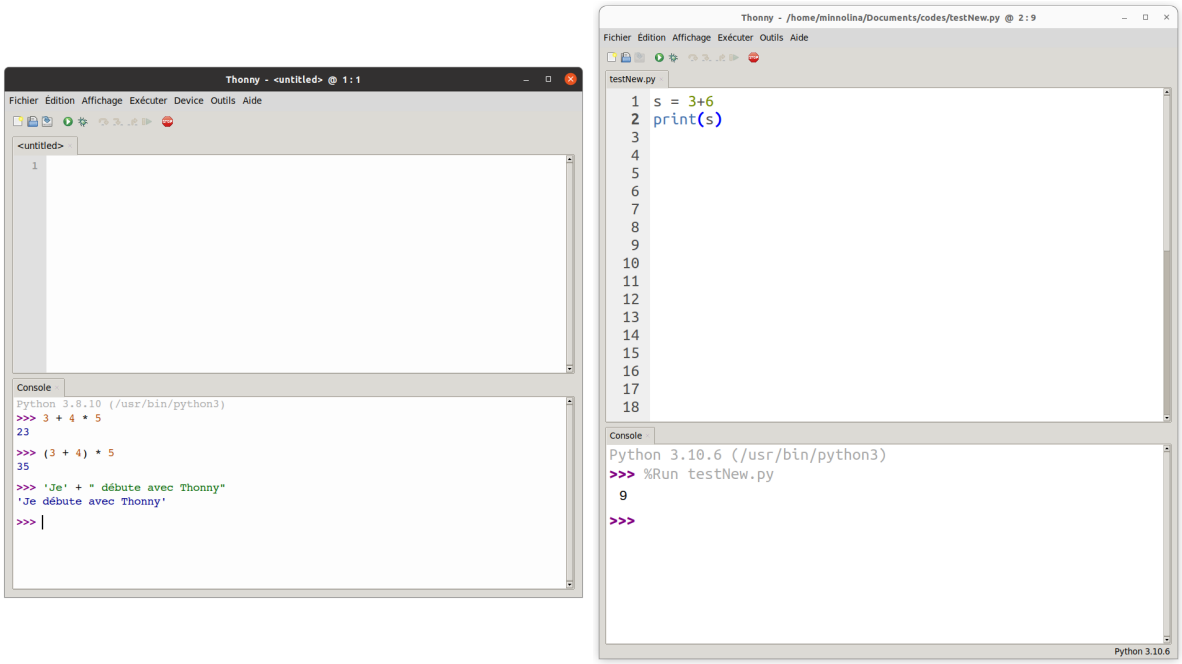
Pour commencer on va apprendre à utiliser Python directement : [4]

- ouvrir un IDE (ou écrire `python3` dans un terminal puis appuyer sur la touche «Entrée») ;
- un invite de commande, composé de trois chevrons (>>>), apparaît : cette marque visuelle indique que Python est prêt à lire une commande. Il suffit de saisir à la suite une instruction puis d'appuyer sur la touche «Entrée». Pour commencer, comme le veux la tradition informatique, on va demander à Python d'afficher les fameux mots «Hello world» : [5]

1. La version 2 de Python est désormais obsolète et a cessé d'être maintenue après le 1er janvier 2020. Dans la mesure du possible évitez de l'utiliser.
2. https://dane.ac-lyon.fr/spip/IMG/scenari/python/co/GC_4-2_prog_Idle.html
3. Voir par exemple https://thonny.org/ et https://realpython.com/python-thonny/. C'est probablement le choix plus simple pour commencer (notamment sous Windows). Il gère aussi très simplement l'installation de modules tels que `scipy`, `sympy` et `matplotlib`.
4. Il ne s'agit pas, pour l'instant, de s'occuper des règles exactes de programmation, mais seulement d'expérimenter le fait d'entrer des commandes dans Python.
5. Pour les curieux qui veulent découvrir mille et mille manière d'afficher "Hello, World!" dans tous les langages informatiques de la Terre, voyez ici http://fr.wikipedia.org/wiki/Liste_de_programme_Hello_world



(a) Lors du lancement d'Idle, l'interpréteur Python démarre par défaut, affichant une console pour l'utilisation de Python en mode interactif à la ligne de commande. Pour ouvrir l'éditeur, cliquez sur `File`, puis sur `New File`, une nouvelle fenêtre s'ouvrira pour le mode script.



(b) Thonny, lors de son ouverture, présente une console (en bas) permettant la saisie de code Python et l'affichage direct des résultats en mode interactif. En haut, se trouve l'éditeur pour la rédaction et la sauvegarde des programmes (scripts).

FIGURE 1.1. – Captures d'écran de deux environnements de développement intégré (IDE).

```
>>> print("Hello world")
Hello world
```

Si l'instruction produit un résultat, il est affiché une fois l'instruction exécutée.

- La console Python fonctionne comme une simple calculatrice : on peut saisir une expression dont la valeur est renvoyée dès qu'on presse la touche «Entrée», par exemple

```
>>> 2*7+8-(100+6)
-84
>>> 2**5
32
```

```
>>> 34/5
6.8
>>> 34//5
6
>>> 34%5
4
```

```
>>> a = 4
>>> b = 10
>>> c = a*b
>>> print(a,b,c)
4 10 40
```

- Pour quitter le mode interactif, il suffit d'exécuter l'instruction `exit()`. Il s'agit de nouveau d'une fonction prédéfinie de Python permettant de quitter l'interpréteur.

Le mode interactif est très pratique pour rapidement tester des instructions et directement voir leurs résultats. Son utilisation reste néanmoins limitée à des programmes de quelques instructions. En effet, devoir à chaque fois retaper toutes les instructions s'avérera vite pénible.

1.1.2. Mode script

En **mode script**, il est nécessaire de préalablement rédiger toutes les instructions du programme dans un fichier texte, puis de l'enregistrer sur l'ordinateur. Habituellement, on utilise l'extension de fichier `.py` pour les fichiers contenant du code Python. Une fois cela accompli, l'interpréteur va lire ce fichier et exécuter son contenu, pas à pas, comme si on les avait entrées une par une dans le mode interactif. Cependant, les résultats intermédiaires des différentes instructions ne sont pas affichés automatiquement ; seuls les affichages explicites (par exemple, avec la fonction `print`) se produiront.

- Tout d'abord, commençons par ouvrir un éditeur. On peut constater qu'il n'y a aucun contenu dans cette nouvelle fenêtre (pas d'en-tête ni de chevrons comme dans l'interpréteur). Cela signifie que ce fichier est exclusivement dédié aux commandes : Python ne fournira pas de réponses tant que nous ne le lui demanderons pas.
- Notre objectif est de conserver les instructions que nous avons essayées dans l'interpréteur. Pour ce faire, nous pouvons les saisir manuellement ou les copier-coller dans ce fichier.
- Enregistrons maintenant le fichier sous le nom `primo.py`. Pour ce faire, nous utilisons la commande «Save» (Sauver) dans le menu «File» (Fichier). Alternativement, le raccourci ctrl + S peut être utilisé.
- Après avoir enregistré le programme, pour l'exécuter et afficher les résultats dans la fenêtre de l'interpréteur, il suffit d'utiliser le raccourci clavier F5 (ou de taper dans le terminal `python primo.py` puis d'appuyer sur la touche «Entrée»).
- Maintenant que le programme est enregistré, nous pouvons le recharger : il suffit de tout fermer, de relancer l'éditeur et d'ouvrir le fichier.

ATTENTION Noter la différence entre l'output produit...

... en mode **interactif** :

```
>>> a = 10
>>> a # valeur de a affichée en mode interactif
10
>>> print("J'ai fini")
J'ai fini
>>> print("a =",a)
a = 10
```

... en mode **script**

```
a = 20
a # valeur non affichée en mode script
print("J'ai fini")
print("a =",a)
```

dont l'output est

```
J'ai fini
a = 20
```

Dans le mode **interactif**, la valeur de la variable `a` est affichée directement tandis que dans le mode **script**, il faut utiliser `print(a)`.

1.2. Commentaires

Il est souvent utile de laisser un commentaire pour expliquer les parties de votre code qui ne sont pas immédiatement évidentes. Le symbole dièse (`#`) indique le début d'un commentaire : tous les caractères entre `#` et la fin de la ligne sont ignorés par l'interpréteur. Les commentaires doivent être utilisés avec modération, maintenus à jour, et indentés de la même façon que la ligne de code suivante. On ne décrit pas le code, on l'explique !

Pour commenter (resp. dé-commenter) plusieurs lignes avec nos IDE, surligner les lignes à traiter et

IDLE appuyer sur les touches ctrl + D (resp. ctrl + ⇧ + D) ;

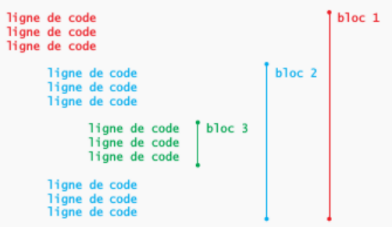
Thonny appuyer sur les touches ctrl + ⇧ + 3 .

1.3. Indentation

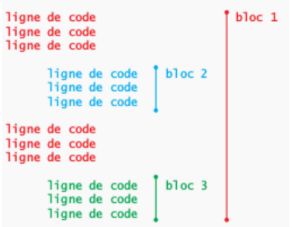
En Python (contrairement aux autres langages) c'est l'indentation (les espaces en début de chaque ligne) qui détermine les blocs d'instructions (boucles, sous-routines, etc.). Cette obligation de coder en utilisant l'indentation permet d'obtenir du code plus lisible et plus propre.

Les lignes dont le code débute au même emplacement (en retrait du même nombre d'espaces depuis la marge gauche) sont regroupées en un bloc. Chaque fois qu'on débute une nouvelle ligne par plus d'espaces que la précédente, on démarre un nouveau sous-bloc qui fait partie du précédent.

Nous regroupons ensemble les instructions dans des blocs, car ils s'associent et doivent être exécutés ensemble :



Si on change le retrait, on crée généralement de nouveaux blocs. Par exemple, trois blocs séparés sont créés simplement en changeant le retrait.



Ici, même si les blocs 2 et 3 ont le même retrait, ils sont considérés comme différents, car un bloc avec un retrait moindre (moins d'espaces) existe entre les deux.

Pour produire une indentation on peut soit appuyer 4 fois sur la barre ␣ ou appuyer une fois sur la touche tabulation ⇆ . L'indentation doit être homogène (soit des espaces, soit des tabulations, mais pas un mélange des deux). Dans ce polycopié on notera chaque tabulation avec une flèche comme suit :

```
for i in range(5):    # aucune tabulation
⟶for j in range(4): # une tabulation
⟶⟶print(i+j)        # deux tabulations
```

Dans la PEP 8, [6] il est recommandé d'utiliser 4 espaces, pour que les développeurs n'obtiennent pas d'erreurs d'«indentation inattendue» lorsqu'ils copient du code. Afin de toujours utiliser cette règle des 4 espaces pour l'indentation, on peut configurer son éditeur de texte pour que, lorsque on presse la touche tabulation, cela ajoute 4 espaces (et non pas un caractère tabulation).

Pour indenter (resp. dés-indenter) plusieurs lignes avec nos IDE, surligner les lignes à traiter et

IDLE appuyer sur les touches ⇆ (resp. ⇧ + ⇆) ;

Thonny appuyer sur la touche ⇆ (resp. ctrl + ⇧ + ⇆).

1.4. Types primitifs

Les différents types primitifs sont :

- `int` : un entier(*Integers* en anglais)
- `float` : un nombre décimal (les virgules flottantes, *Float* en anglais)
- `complex` : un nombre complexe

6. PEP est l'abréviation de *Python Enhancement Proposal* (proposition d'amélioration de Python). Les PEP sont des documents de conception pour la communauté Python. Elles décrivent des nouvelles fonctionnalités pour Python, ses processus ou son environnement. La PEP qui sert de guide de style pour Python est la PEP 8. Il s'agit d'une longue liste de pratiques suggérées pour les développeurs Python. Créée en 2001, elle a été écrite par Guido VAN ROSSUM, Barry WARSAW et Nick COGHLAN, et elle est mise à jour régulièrement pour refléter les développements du langage. Pour vérifier tout élément spécifique, vous pouvez consulter le document officiel sur le site de Python https://peps.python.org/pep-0008/.

- `bool` : un booléen (`True` ou `False`)

On peut identifier le type d'une variable en utilisant `type()` :

```
>>> type(10)
<class 'int'>
```

```
>>> type(3.5)
<class 'float'>
```

```
>>> type(8+2j)
<class 'complex'>
```

```
>>> type(3>5)
<class 'bool'>
```

Noter que les nombre `100` et `100.0` ont l'air de se ressembler, mais dans Python, l'un est un entier et l'autre est un nombre à virgule flottante.

```
>>> type(100), type(100.0)
(<class 'int'>, <class 'float'>)
```

Comment fait Python pour savoir qu'un nombre est entier ? Comme le langage est faiblement typé, il doit le deviner. Le critère est simple : pour qu'un nombre soit entier, il ne doit pas avoir de virgule (représentée dans Python par un point décimal).

Certains calculs devant donner un résultat entier ne le font pas toujours en Python. Par exemple, alors que $\frac{10}{5} \in \mathbb{N}$, Python considère ce nombre comme un réel (non entier) :

```
>>> 10/5, type(10/5)
(2.0, <class 'float'>)
```

On peut convertir le type de nombre en un autre.

- Les opérations telles que l'addition ou la soustraction convertissent implicitement (automatiquement) les nombres entiers en nombres flottants, si l'un des opérandes est un nombre flottant. Par exemple,

  ```
  >>> type(1 + 2.0)
  <class 'float'>
  ```

- Nous pouvons également utiliser des fonctions intégrées telles que `int()`, `float()` et `complex()` pour convertir les types de manière explicite.

  ```
  >>> type(2.3), type(int(2.3))
  (<class 'float'>, <class 'int'>)
  ```

  Lors de la conversion de flottant en entier, le nombre est tronqué (certains chiffres sont supprimés). Inversement, lors de la conversion d'un entier en un flottant, `.0` est postfixé au nombre.

Il existe ensuite cinq structures de référence : les chaînes de caractères `str`, les listes `list`, les tuples `tuple`, les dictionnaires `dict` et les ensembles `set`. Ces structures sont en fait des objets qui peuvent contenir d'autres objets. On les verra au prochaine chapitre.

```
>>> type( "Ciao !" )
<class 'str'>
>>> type( [ 10 , "Ciao !" , 3.14 , True ] )
<class 'list'>
>>> type( ( 10 , "Ciao !" , 3.14 , True ) )
<class 'tuple'>

>>> type( { "a":10 , "b":5 } )
<class 'dict'>
>>> type( set( [10 , "Ciao !" , 10 , True] ) )
<class 'set'>
```

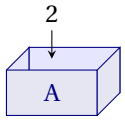
1.5. Variables et affectation

Une variable est une sorte de boîte contenant un objet, par exemple une valeur. Cette boîte est stockée dans un gigantesque entrepôt (la mémoire de l'ordinateur). Son emplacement est très précisément répertorié (adresse mémoire). À chaque boîte est attribué un nom afin de facilement l'identifier. On va également avoir besoin d'appliquer différentes opérations sur ces boîtes comme les vider, modifier le contenu, transférer le contenu de l'une à l'autre, etc.

La session interactive suivante avec l'INTERPRÉTEUR Python illustre ce propos (>>> est le prompt) :

```
>>> A = 2
>>> print(A)
2
```

L'affectation `A = 2` crée une association entre le nom `A` et le nombre entier 2 : la boîte de nom `A` contient la valeur 2.



Il faut bien prendre garde au fait que **l'instruction d'affectation** (=) **n'a pas la même signification que le symbole d'égalité** (=) **en mathématiques** (ceci explique pourquoi l'affectation de 2 à `A`, qu'en Python s'écrit `A = 2`, en algorithmique se note souvent $A \leftarrow 2$).

Le nom d'une variable doit toujours représenter son contenu, avec des noms clairs et précis. Avec des noms bien choisis, on comprend tout de suite ce que calcule le code suivant :

```
base = 8
hauteur = 3
aire = base * hauteur / 2
print(aire)
```

Voici quelques recommandations générales pour choisir un nom :

- **Utilisez des noms descriptifs.** Les noms de variables descriptifs et spécifiques simplifient la vie et facilitent la lecture et la modification du code. Le code est lu bien plus souvent qu'il n'est écrit : évitons donc de nous compliquer la tâche avec des abréviations, des mots hachés et des variables à une lettre. Le temps gagné sur le moment ne rattrapera jamais le temps perdu à relire ; et la qualité générale du code s'en ressentira.
- Les noms de variables sont sensibles à la casse : `age`, `Age` et `AGE` sont trois variables différentes.
- Utilisez uniquement des caractères alphanumériques et des tirets bas «_» (appelé *underscore* en anglais) et pas d'accents. Par ailleurs, un nom de variable ne peut pas utiliser d'espace, ne doit pas débuter par un chiffre et il n'est pas recommandé de le faire débuter par le caractère _ (sauf cas très particuliers).

  Le style recommandé selon le PEP 8 pour nommer les variables et les fonctions en Python est la **convention *snake_case***:
  - les noms de variables, de fonctions et de modules doivent être en minuscules, avec éventuellement des underscores (tirets bas) pour séparer les différents mots dans le nom :

    ```
    name = "Jeanne"
    fuel_level = 100
    famous_singers = ["Céline Dion", "Michael Jackson", "Edith Piaf"]
    ```
  - les constantes sont écrites en majuscules, avec éventuellement des underscores :

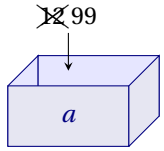
    ```
    DAYS_PER_WEEK = 7
    PERSONAL_EMAIL = "myemail@email.com"
    ```

    En Python on ne peut pas créer des véritables constante dont la valeur ne peut pas être modifiée. Cependant, le simple fait de l'écrire TOUT_EN_MAJUSCULES signale aux autres développeurs de ne pas écrire du code qui la modifie.
- Il faut absolument **éviter d'utiliser un mot «réservé» ou le nom d'une fonction déjà définie** par Python comme nom de variable ou de fonction.
  - *Liste des mots-clés réservés prédéfinis :* and, as, assert, async, await, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, while, with, yield.
  - *Liste des fonctions prédéfinies :* abs, aiter, all, anext, any, ArithmeticError, ascii, AssertionError, AttributeError, BaseException, BaseExceptionGroup, bin, BlockingIOError, bool, breakpoint, BrokenPipeError, BufferError, bytearray, bytes, BytesWarning, callable, ChildProcessError, chr, classmethod, compile, complex, ConnectionAbortedError, ConnectionError, ConnectionRefusedError, ConnectionResetError, copyright, credits, delattr, DeprecationWarning, dict, dir, divmod, Ellipsis, EncodingWarning, enumerate, EnvironmentError, EOFError, eval, Exception, ExceptionGroup, exec, exit, False, FileExistsError, FileNotFoundError, filter, float, FloatingPointError, format, frozenset, FutureWarning, GeneratorExit, getattr, globals, hasattr, hash, help, hex, id, ImportError, ImportWarning, IndentationError, IndexError, input, int, InterruptedError, IOError, IsADirectoryError, isinstance, issubclass, iter, KeyboardInterrupt, KeyError, len, license, list, locals, LookupError, map, max, MemoryError, memoryview, min, ModuleNotFoundError, NameError, next, None, NotADirectoryError, NotImplemented, NotImplementedError, object, oct, open, ord, OSError, OverflowError, PendingDeprecationWarning, PermissionError, pow, print, ProcessLookupError, property, quit, range, RecursionError, ReferenceError, repr, ResourceWarning, reversed, round, RuntimeError, RuntimeWarning, set, setattr, slice, sorted, staticmethod, StopAsyncIteration, StopIteration, str, sum, super, SyntaxError, SyntaxWarning, SystemError, SystemExit, TabError, TimeoutError, True, tuple, type, TypeError, UnboundLocalError, UnicodeDecodeError, UnicodeEncodeError, UnicodeError, UnicodeTranslateError, UnicodeWarning, UserWarning, ValueError, vars, Warning, ZeroDivisionError, zip.

Une fois une variable initialisée, on peut modifier sa valeur en utilisant de nouveau l'opérateur d'affectation (=). La valeur actuelle de la variable est remplacée par la nouvelle valeur qu'on lui affecte. Le type de la variable va changer de lui-même en fonction de la valeur stockée dedans.

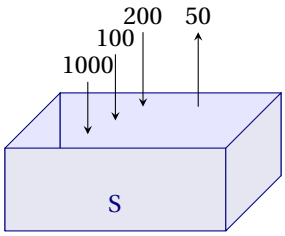
Dans l'exemple ci-contre, on initialise une variable à la valeur 12 et on remplace ensuite sa valeur par 99 :

```
>>> a = 12
>>> a = 99
>>> print(a)
99
```



Un autre exemple (on part d'une somme S = 1000, puis on lui ajoute 100, puis 200, puis on enlève 50). Il faut comprendre l'instruction S=S+100 comme ceci : «je prends le contenu de la boîte S, je rajoute 100, je remets tout dans la même boîte».

```
>>> S = 1000
>>> S = S + 100
>>> S = S + 200
>>> S = S - 50
>>> print(S)
1250
```



Remarque 1
Il est important de garder le contrôle des valeurs des variables. En particulier, lors de l'apprentissage, il est important de s'assurer que le code fait ce qu'il est censé faire, et ce chaque fois que nous créons ou modifions une variable. L'affichage avec la fonction `print()` est un moyen facile de vérifier que les modifications de variables correspondent à nos intentions.

On souhaite parfois conserver en mémoire le résultat de l'évaluation d'une expression arithmétique en vue de l'utiliser plus tard. Par exemple, si on recherche les solutions de l'équation ax^2+bx+c, on doit mémoriser la valeur du discriminant pour pouvoir calculer les valeurs des deux racines réelles distinctes, lorsqu'il est strictement positif.

```
>>> a = 1
>>> b = 2
>>> c = 4
>>> # on peut aussi écrire les instructions sur une seule ligne
>>> # a = 1; b = 2; c = 4;
>>> delta = b**2-4*a*c
>>> print(delta)
-12
```

Avant de pouvoir accéder au contenu d'une variable, il faut qu'elle soit initialisée, c'est-à-dire qu'elle doit posséder une valeur. Si on tente d'utiliser une variable non initialisée, l'exécution du programme va s'arrêter et l'interpréteur Python va produire une **erreur** d'exécution. Voyons cela avec l'exemple de programme suivant :

```
>>> a = 178
>>> print('Sa taille est :')
Sa taille est :
>>> print(toto)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'toto' is not defined
```

L'avant-dernière ligne reprend l'instruction qui a causé l'erreur d'exécution (à savoir `print(toto)` dans notre cas). La dernière ligne fournit une explication sur la cause de l'erreur (celle qui commence par `NameError`). Dans cet exemple, elle indique que le nom `toto` n'est pas défini, c'est-à-dire qu'il ne correspond pas à une variable initialisée.

Remarque 2 (Les messages d'erreur)
Faire des erreurs en programmant est normal : il faut apprendre à lire les messages d'erreur pour les corriger efficacement. En Python, **ils se lisent de bas en haut** : la dernière ligne indique le **type d'erreur**, l'avant-dernière la **ligne concernée**. Le reste retrace la "pile d'appels" ayant mené à l'erreur. Attention : ce n'est pas toujours la ligne signalée qu'il faut corriger, mais parfois celle qui l'a précédée.

Il arrive aussi qu'un programme ne produise *aucune erreur*, tout en donnant un résultat inattendu. Dans ce cas, c'est votre raisonnement qu'il faut examiner : **l'ordinateur exécute exactement ce qu'on lui demande, pas ce qu'on pense lui avoir demandé.**

Affectations multiples en parallèle On peut aussi effectuer des affectations parallèles :[7]

7. Il s'agit de tuples définies sans écrire explicitent les parenthèses.

```
>>> x = y = z = 50
>>> print(x); print(y); print(z)
50
50
50
```

```
>>> a, b = 128, 256
>>> print(a); print(b)
128
256
```

Cette syntaxe permet d'ÉCHANGER très facilement le contenu de deux variables :

```
>>> a, b = 10, 5
>>> a, b = b, a
>>> print(a)
5
>>> print(b)
10
```

ATTENTION Noter la différence lorsqu'on écrit les instructions suivantes :

```
>>> a = 10; b = 5
>>> a = b
>>> b = a
>>> print(a); print(b)
5
5
```

```
>>> a = 10; b = 5
>>> tmp = b
>>> b = a
>>> a = tmp
>>> print(a); print(b)
5
10
```

```
>>> a = 10; b = 5
>>> a = a+b
>>> b = a-b
>>> a = a-b
>>> print(a); print(b)
5
10
```

La première approche n'échange pas le contenu des deux variables. En effet, la première instruction `a = b` a pour effet de mettre le contenu de la variable `b` dans la variable `a` et d'y effacer le contenu précédent. Par conséquent, avant de mettre le contenu de `b` dans `a`, il faut sauvegarder dans une troisième variable le contenu de `a` pour pouvoir ensuite le récupérer afin de le mettre dans `b`. C'est le rôle de `tmp` dans le deuxième exemple. Le troisième exemple exploite une propriété mathématique mais risque de donner quelque surprise si les nombres à échanger ne sont pas des entiers, notamment si leur différence est très grande (voir l'annexe A).

1.6. Nombres et opérations arithmétiques

Python comprend trois types numériques fondamentaux :

- Le type `int` (entiers) peut représenter n'importe quel nombre entier, quel que soit sa taille.
- Le type `float` (décimaux) permet de représenter des nombres avec une partie décimale (comme 3.14 ou 2.1e − 23), avec jusqu'à 15 chiffres significatifs, compris entre 10^{-308} et 10^{308}. À noter : pour écrire des nombres décimaux, le point est utilisé comme séparateur. La valeur spéciale `math.inf` représente l'infini (voir le chapitre sur les modules).
- Le type `complex` (nombres complexes) permet de représenter des nombres complexes, utilisant `j` pour la partie imaginaire (par exemple `3.1+5.2j`). Il suffit de coller directement la partie imaginaire d'un nombre complexe à `j`, sans utiliser de `*`. Par exemple, `2j` représente un nombre complexe, tandis que `2 * j` représente le produit de 2 par une variable (qui doit être définie) appelée `j`.

Dans Python on a les opérations arithmétiques usuelles :

`+`	Addition	`**`	Puissance, Exponentiation
`-`	Soustraction		
`*`	Multiplication	`//`	Division entière, Quotient de la division euclidienne
`/`	Division	`%`	Modulo, Reste de la division euclidienne

Un exemple :

```
>>> a = 100
>>> b = 17
>>> c = a*(a-b)
>>> print(a,b,c)
100 17 8300
```

```
>>> a = 13
>>> b = a//2
>>> print(a,b)
13 6
```

```
>>> a = 2
>>> b = a**0.5 # ∛x = x^(1/n)
>>> print(a,b)
2 1.4142135623730951
```

Opérateurs augmentés. Lorsqu'on veut changer une variable en modifiant la valeur initiale via un opérateur basique, on peut utiliser une version plus courte. Il s'agit des **opérateurs augmentés** :

`a += b` équivaut à a = a+b	`a -= b` équivaut à a = a-b	`a *= b` équivaut à a = a*b
`a /= b` équivaut à a = a/b	`a **= b` équivaut à a = a**b	`a %= b` équivaut à a = a%b

Priorités. Les opérateurs arithmétiques possèdent chacun une **priorité** qui définit dans quel ordre les opérations sont effectuées. Par exemple, lorsqu'on écrit `1 + 2 * 3`, la multiplication va se faire avant l'addition. Le calcul qui sera effectué est donc `1 + (2 * 3)`. Dans l'ordre, l'opérateur d'exponentiation est le premier exécuté, viennent ensuite les opérateurs *, /, // et %, et enfin les opérateurs + et -. Lorsqu'une expression contient plusieurs opérations de même priorité, ils sont évalués de gauche à droite. Ainsi, lorsqu'on écrit `1 - 2 - 3`, le calcul qui sera effectué est `(1 - 2) - 3`. En cas de doutes, vous pouvez toujours **utiliser des parenthèses** pour rendre explicite l'ordre d'évaluation de vos expressions arithmétiques.

Typecasting. Si le résultat d'une opération entre deux entiers est censé être un nombre décimal, Python va automatiquement le convertir en `float`. De plus, **la division (même si le résultat est censé être entier) renverra forcément un** `float` également.

```
>>> a = 100
>>> print(type(a))
<class 'int'>
```

```
>>> b = 10.0
>>> print(type(b))
<class 'float'>
```

```
>>> c = a-b
>>> print(type(c))
<class 'float'>
```

```
>>> d = a/2
>>> print(type(d))
<class 'float'>
```

On peut forcer la conversion d'une variable dans un type bien défini. Ceci est appelé du *typecasting*, car en faisant ainsi, on remodèle (*cast* en anglais) le type d'une variable. Pour ce faire, on a besoin des fonctions correspondantes

1. `int(x)` pour convertir x en un entier,
2. `float(n)` pour convertir n en un nombre décimal.

```
>>> a = 100
>>> b = 10.0
```

```
>>> a_bis = float(a)
>>> b_bis = int(b)
```

```
>>> print(type(a_bis))
<class 'float'>
>>> print(type(b_bis))
<class 'int'>
```

Division entière. La division entière d'un entier D par un entier d fournit deux résultats : un quotient q et un reste r tels que $D = qd + r$ avec $r < d$. Par exemple, 17 divisé par 5 donne un quotient de 3 et un reste de 2, car $17 = 3 \times 5 + 2$. Ces notions sont utiles dans de nombreuses situations : déterminer si un entier est pair ou impair (reste nul ou égal à 1 lors de la division par 2), ou encore convertir des durées. Ainsi, 175 secondes correspondent à 2 minutes et 55 secondes.
En Python, la division entière se note // et le reste %. La fonction `divmod` renvoie directement le quotient et le reste.

```
>>> print(9//4)
2
>>> print(9%4)
1
>>> print(divmod(9,4))
(2, 1)
```

```
>>> # Conversion secondes → minutes et secondes
>>> print(175//60)
2
>>> print(175%60)
55
```

1.7. Type booléen, Opérateurs de comparaison et connecteurs logiques

Le type booléen est l'un des types de données intégrés fournis par Python, qui représente l'une des deux valeurs, à savoir `True` (vrai) ou `False` (faux). Les opérateurs de comparaison renvoient `True` si la condition est vérifiée, `False` sinon. Une condition est une instruction qui compare des choses et indique si le critère de comparaison est vrai (booléen `True`) ou faux (booléen `False`). Ainsi, "`age>18`" est une condition et revient à demander "la valeur de la variable `age` est-elle plus grande que `18`?". Pour combiner des conditions complexes (par exemple $x > -2$ et $x^2 < 5$), on peut combiner des variables booléennes en utilisant les connecteurs logiques. Ces opérateurs et connecteurs s'écrivent comme suit :

On écrit	<	>	<=	>=	==	!=	in	and	or	not
Ça signifie	$<$	$>$	$\leq$	$\geq$	$=$	$\neq$	$\in$	et	ou	non

Attention (= *vs* ==) Bien distinguer l'instruction d'affectation = du symbole de comparaison ==.

Remarque 3
Dans une condition formée de la conjonction (avec `and`) de plusieurs prédicats, dès qu'un prédicat est faux, les suivants ne sont même pas évalués. L'ordre des prédicats est donc important.

Deux nombres de type différents (entier, à virgule flottante, etc.) sont convertis en un type commun avant de faire la comparaison. Dans tous les autres cas, deux objets de type différents sont considérés non égaux. Voici quelques exemples :

```
>>> a = 2     # Integer
>>> b = 1.99  # Floating
>>> c = '2'   # String
```

```
>>> print(a>b)
True
>>> print(a==c)
False
```

```
>>> print( (a>b) and (a==c) )
False
>>> print( (a>b) or (a==c) )
True
```

La PEP 8 recommande d'entourer l'instruction d'affectation = et les opérateurs (+, -, /, *, ==, !=, >=, not, in, and, or...) d'un espace avant et d'un espace après.

Code recommandé :

```
ma_variable = 3 + 7
mon_texte = "souris"
mon_texte == ma_variable
```

Code non recommandé :

```
ma_variable=3+7
mon_texte="souris"
mon_texte== ma_variable
```

Remarque 4
Nous utiliserons les opérateurs `and` et `or` seulement avec des booléens mais attention, ils peuvent être utilisés avec n'importe quel objets.

- `and` renvoie le premier élément s'il est `False` ou `0` ou `None` ou `""` etc., sinon il renvoie le deuxième ;
- `or` renvoie le premier élément s'il est `True` ou `!=0` ou non vide etc., sinon il renvoie le deuxième.

```
>>> print( 1 and 2 )
2
```

```
>>> print( 0 and 2 )
0
```

```
>>> print( 1 or 2 )
1
```

```
>>> print( 0 or 2 )
2
```

Attention (Comparaison de nombres flottants) Lorsqu'on écrit $a = x$, où x est un nombre réel, la valeur a stockée en machine est en général une approximation de x (plus rarement la valeur exacte). Cette approximation est représentée sous la forme d'un nombre flottant (type `float` en Python). Ainsi, contrairement aux calculs sur les réels mathématiques, les opérations et les comparaisons effectuées sur leur représentation en machine sont approchées. Il peut alors arriver que deux réels égaux aient des représentations flottantes différentes, ou inversement. Par exemple :

```
>>> 1/3 == 0.33333333333333333333  # réponse attendue : False
True
>>> 0.1+0.2 == 0.3  # réponse attendue : True
False
```

Ces résultats sont dus aux erreurs d'arrondi introduites lors de la représentation binaire des nombres flottants.

Pour cette raison, lors de comparaisons entre nombres flottants, on ne teste **jamais l'égalité exacte**, mais la **proximité** des valeurs à une tolérance près :

```
>>> delta = 1.e-14  # précision choisie
>>> x = 1/10
>>> y = 0.1
>>> abs(x - y) < delta
True
```

Ce test indique que les deux valeurs sont suffisamment proches, avec une tolérance de 10^{-14}, et doivent être considérées comme égales dans un contexte numérique.

1.8. Exercices

Attention

Quand on apprend à développer, il est fondamental de saisir soi-même chaque commande plutôt que de copier/coller. La saisie **active** favorise la mémorisation. De plus, les doigts intègrent des schémas de frappe : par exemple, `print()` et ses parenthèses deviennent un réflexe.

Exercice 1.1 (Devine le résultat – affectations)

Prédire le résultat de chacune des instructions suivantes puis vérifier-les dans l'interpréteur Python :

```
a = 1
b = 100
b = a
a = b
print(a, b)
```

```
a = 1
b = 100
a = b
b = a
print(a, b)
```

```
a = 1
b = 100
a,b = b,a
print(a, b)
```

Correction

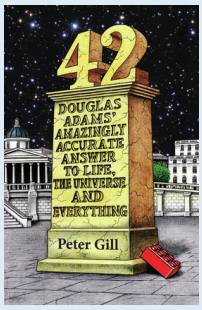
Premier script : 1 1 Deuxième script : 100 100 Dernier script : 100 1

Pour visualiser le contenu de chaque variable pas à pas on peut utiliser pythontutor.com

Exercice 1.2 (Devine le résultat – affectations)

Le Guide du voyageur galactique (titre original : *The Hitchhiker's Guide to the Galaxy*, abrégé notamment en *H2G2*) est une œuvre de science-fiction humoristique de l'écrivain britannique Douglas ADAMS. Selon ce guide, des chercheurs d'un peuple hyper-intelligent et pan-dimensionnel construisirent le deuxième plus grand ordinateur de tous les temps, *Pensées profondes*, pour calculer la réponse à la grande question sur la vie, l'Univers et le reste. Après sept millions et demi d'années à réfléchir à la question, Pensées profondes fournit enfin la réponse. Pour la découvrir, prédire le résultat de chacune des instructions suivantes :

```
Hitchhiker, Guide, to, the, Galaxy = 0, 1, 2, 3, 4
The = the-to
Answer = Galaxy**0.5
to = to-1
the = The
Ultimate = 3*Guide
Question = 0*Hitchhiker + 0*Guide + 0*to + 0*the + Galaxy / 4
of = Galaxy**Hitchhiker
Life = Galaxy*the + Answer*Guide + Ultimate - 2
number = The*Answer*to*the*Ultimate*Question*of*Life
number = int(number)
```



Correction

Vous pouvez visualiser les affectations pas à pas sur pythontutor.com

```
Hitchhiker, Guide, to, the, Galaxy = 0, 1, 2, 3, 4
The = the-to          # 1
Answer = Galaxy**0.5  # 2
to = to-1             # 1
the = The             # 1
Ultimate = 3*Guide    # 3
Question = 0*Hitchhiker + 0*Guide + 0*to + 0*the + Galaxy / 4 # 1.0
of = Galaxy**Hitchhiker # 4^0 = 1
Life = Galaxy*the + Answer*Guide + Ultimate - 2 # 4×1+2×1+3−2=7
number = The*Answer*to*the*Ultimate*Question*of*Life # 1×2×1×1×3×1.0×1×7=42.0
number = int(number) # 42
print(number)
```

```
42
```

Exercice 1.3 (Séquentialité)

On donne les lignes de code suivantes, désordonnées. Réécrire le script dans le bon ordre afin qu'il affiche le nombre 46. Et que faut-il modifier pour afficher 49 à la place ?

```
y = y-1      # instr. a
y = 2*x      # instr. b
print(x)     # instr. c
x = x+3*y    # instr. d
x = 7        # instr. e
```

Correction

On ne peut utiliser une variable que si elle a déjà été initialisée. Ainsi : l'instruction e (affectation de x) doit venir avant toutes celles qui utilisent x ; l'instruction b (affectation de y) doit précéder celles qui utilisent y, c'est-à-dire a et d ; enfin, l'affichage c doit venir en dernier. L'ordre correct est donc

```
x = 7        # x ← 7
y = 2*x      # y ← 2x = 14
y = y-1      # y ← y − 1 = 14 − 1 = 13  (on pourrait écrire  y -= 1)
x = x+3*y    # x ← x + 3y = 7 + 3 × 13 = 46  (on pourrait écrire x += 3*y)
print(x)     # affiche 46
```

Pour obtenir 49, il suffit de supprimer la ligne `y = y - 1`, car alors $y = 14$ et $x = 7 + 3 \times 14 = 49$.

Exercice 1.4 (Devine le résultat – logique)

Quel résultat donnent les lignes de code suivantes ?

Cas 1
```
x = 3
print( x == 3 )
print( x != 3 )
print( x >= 4 )
print( not(x<4) )
```

Cas 2
```
a = 7
print( a>5 and a<10 )
print( 5<a<10 )
```

Cas 3
```
a = 15
print( a>5 or a<10 )
print( a<5 or a>10 )
```

Cas 4
```
a, b, c = 1, 10, 100
print( a<b<c )
print( a>b>c )
```

Correction

- Cas 1 : True False False False
- Cas 2 : True True
- Cas 3 : True True
- Cas 4 : True False

Exercice 1.5 (Mode interactif)

Lancer le logiciel **Thonny** à partir du menu "Applications" (sur la page Moodle vous trouverez des indications sur la configuration et des conseils d'utilisation). Dans la partie inférieure de la fenêtre, on voit apparaître la version de python et les trois chevrons `>>>` indiquant qu'on a lancé l'interpréteur. Essayer les instructions suivantes et noter la priorité des instructions.

```
13/2                    42                      .6*9+.6                 13%2
13/2.                   2*12                    round(.6*9+.6)          13//2
                        2**10                                           divmod(13,2)
                        ((19.9*1.2)-5)/4                                exit()
```

Correction

On constate qu'on peut utiliser l'interpréteur Python en mode interactif comme une calculatrice. En effet, si vous y tapez une expression mathématique, cette dernière est évaluée et son résultat affiché comme résultat intermédiaire (autrement dit, il n'est pas nécessaire d'utiliser `print`, ce qui n'est pas le cas avec le mode script).

```
>>> 13/2
6.5
>>> 13/2.
6.5
```

```
>>> 2*12
24
>>> 2**10
1024
>>> ((19.9*1.2)-5)/4
4.72
```

```
>>> .6*9+.6
5.999999999999999
>>> round(.6*9+.6)
6
```

```
>>> 13%2
1
>>> 13//2
6
>>> divmod(13,2)
(6, 1)
```

Exercice 1.6 (CodEx: Autour des booléens)

Un objet de type `bool` ne peut avoir que l'une des deux valeurs `True` ou `False`, appelées valeurs booléennes. Ces valeurs permettent de donner une valeur de vérité à une proposition: elle est vraie ou fausse. Par exemple, la proposition «3 est strictement inférieur à 4» est vraie. Traduite en Python, elle devient `3 < 4` et est évaluée à `True`. À l'inverse, la proposition «le caractère "a" est présent dans le mot "chien"» est fausse: l'expression `"a" in "chien"` est évaluée par Python à `False`.

Complétez le code ci-dessous en saisissant les propositions demandées, et non leur valeur de vérité. Le résultat de l'évaluation de chaque proposition par Python est affecté à une variable (`prop_1`, `prop_2`...).

1. «3 est strictement inférieur à 4»
2. «3 est supérieur ou égal à 4»
3. «Le caractère 'n' est dans la chaîne 'bonjour'»
4. «Le calcul 3 + 5 * 2 vaut 16»
5. «5 est un nombre pair»
6. «12 est dans la liste list(range(12))»
7. «'ju' n'est pas dans 'bonjour'»
8. «3 est égal à 1 + 2 et 'a' est dans 'Boole'»
9. «3 est égal à 1 + 2 ou 'a' est dans 'Boole'»
10. «6 est un nombre pair et un multiple de 3»
11. «648 est dans les tables de 2, de 3 et de 4»
12. «25 est strictement compris entre 24 et 26»
13. «'a' est dans 'hello' ou dans 'hallo'»
14. «8 est égal à 4 * 2 et différent de 9 - 1»
15. «7 est entre 1 et 5 (inclus l'un et l'autre) ou strictement supérieur à 6»
16. «5 est strictement compris entre 7 et 8 ou (8 est strictement inférieur à 9)»
17. «7 est strictement inférieur à 5 et (8 est strictement supérieur à 5 ou strictement inférieur à 9)»

Source: https://codex.forge.apps.education.fr/exercices/autour_des_booleens/

Correction

```
>>> # 1 : "3 est strictement inférieur à 4"
>>> 3 < 4
True
>>> # 2 : "3 est supérieur ou égal à 4"
>>> 3 >= 4
False
>>> # 3 : "le caractère 'n' est dans la chaine 'bonjour'"
>>> 'n' in 'bonjour'
True
>>> # 4 : "Le calcul 3 + 5 * 2 vaut 16"
>>> (3 + 5 * 2) == 16
False
>>> # 5 : "5 est un nombre pair"
>>> 5 % 2 == 0
False
>>> # 6 : "12 est dans la liste list(range(12))"
>>> 12 in list(range(12))
False
>>> # 7 : "'ju' n'est pas dans 'bonjour'"
>>> 'ju' not in 'bonjour'
True
>>> # 8 : "3 est égal à 1 + 2 et 'a' est dans 'Boole'"
>>> (3 == 1 + 2) and ('a' in 'Boole')
False
>>> # 9 : "3 est égal à 1 + 2 ou 'a' est dans 'Boole'"
```

```
>>> (3 == 1 + 2) or ('a' in 'Boole')
True
>>> # 10 : "6 est un nombre pair et un multiple de 3"
>>> (6 % 2 == 0) and (6 % 3 == 0)
True
>>> # 11 : "648 est dans les tables de 2, de 3 et de 4"
>>> (648 % 2 == 0) and (648 % 3 == 0) and (648 % 4 == 0)
True
>>> # 12 : "25 est strictement compris entre 24 et 26"
>>> 24 < 25 < 26
True
>>> # 13 : "'a' est dans 'hello' ou dans 'hallo'"
>>> ('a' in 'hello') or ('a' in 'hallo')
True
>>> # 14 : "8 est égal à 4 * 2 et différent de 9 - 1"
>>> (8 == 4 * 2) and (8 != 9 - 1)
False
>>> # 15 : "7 est entre 1 et 5 (inclus l'un et l'autre) ou strictement supérieur à 6"
>>> (1 <= 7 <= 5) or (7 > 6)
True
>>> # 16 : "5 est strictement compris entre 7 et 8 ou (8 est strictement inférieur à 9)"
>>> (7 < 5 < 8) or (8 < 9)
True
>>> # 17 : "7 est strictement inférieur à 5 et (8 est strictement supérieur à 5 ou strictement inférieur
↪ à 9)"
>>> (7 < 5) and ((8 > 5) or (8 < 9))
False
```

Exercice 1.7 (Mode script)

Lancer le logiciel **Thonny**. Dans la partie supérieure se trouve l'éditeur de texte intégré. Y saisir les lignes suivantes

```
a, b = 5, 6
c = a + b
print("c =", c)
```

Sauvegarder le fichier (par exemple sous le nom `exo_1_12.py`) puis appuyer sur la touche F5 qui exécute le script.

Correction

L'exécution affiche c = 11 .

Le mode interactif de Python ne permet pas de développer des programmes longs : il faut tout retaper à chaque fois. Le **mode script**, lui, permet d'enregistrer le code dans un fichier texte (extension `.py`), que l'on peut modifier et relancer facilement. C'est le mode utilisé pour écrire de vrais programmes.

- Éditeur : nous avons utilisé celui intégré à Thonny, mais tout éditeur de texte convient. Par exemple, ouvrir le fichier dans *Geany* (menu "Applications" puis "Programmation"), ajouter la ligne `print("Une ligne de plus")`, sauvegarder, puis relancer le script dans Thonny : la modification est bien prise en compte.
- Interpréteur : Thonny exécute le code avec l'interpréteur Python. On peut aussi lancer le script depuis un terminal (menu "Applications" puis "Outils Système") : `python3 exo_1_12.py` (assurez-vous d'être dans le bon dossier ; utilisez `ls` et `cd` pour naviguer). Le résultat s'affiche alors directement dans le terminal.

Exercice Bonus 1.8 (Rendre un script exécutable : la ligne *shebang*)

1. Ouvrir le fichier `exo_1_12.py` dans un éditeur de texte pur.
2. Ajouter au tout début du fichier

```
#! /usr/bin/env python3
# coding: utf-8
```

3. Sauvegarder le fichier.
4. Dans un terminal, rendre le script exécutable : `chmod +x exo_1_12.py`

5. Puis l'exécuter directement : `./exo_1_12.py`

Il n'est alors plus nécessaire d'écrire `python3 exo_1_12.py`.

Correction

Il est recommandé d'ajouter ces deux lignes en tête de tous vos scripts. La première ligne, appelée *shebang*, indique au système d'utiliser l'interpréteur `python3` pour exécuter le script. Elle garantit le bon fonctionnement, même si plusieurs versions de Python coexistent sur la machine. La seconde précise l'encodage du fichier source (`utf-8`), nécessaire pour gérer correctement les caractères accentués. En Unix, le *shebang* doit toujours être placé en tout début de fichier.

Attention

À partir de ce moment, pour chaque exercice **on écrira les instructions dans un fichier script** nommé par exemple `exo_1_11.py` (sans espaces et sans points sauf pour l'extension .py). On pourra bien-sûr utiliser le mode interactif pour simplement vérifier une commande mais chaque exercice devra in fine être résolu dans un fichier de script. Ne pas oublier la différence entre l'output produit en mode *interactif* et l'output produit en mode *script* (*cf.* page 13) ; dans le doute, utiliser toujours la fonction `print`).

Exercice 1.9 (Évaluation numérique d'une fraction)

Pour $x = 1$ et $h = \frac{1}{10}$, calculer et afficher la quantité

$$\frac{(x+h)^2 - (x-h)^2}{2h}.$$

Reprendre ce calcul pour des valeurs de plus en plus petites de h (par exemple $h = 10^{-2}, 10^{-3}, 10^{-4}$).

Remarque : on reconnaît une approximation de la dérivée de la fonction $f(x) = x^2$ au point x. Or $f'(x) = 2x$, donc lorsque $h \to 0$, cette expression converge vers $2x$. Pour $x = 1$, la valeur limite attendue est donc 2.

Correction

```
h = 1/10 # ou 10**(-1) ou 1.e-1
x = 1
y = ( (x+h)**2 - (x-h)**2 ) / (2*h)
print(y)
```

```
2.0000000000000004
```

Mathématiquement, on a $\lim_{h\to 0} \frac{(x+h)^2-(x-h)^2}{2h} = 2x$ pour tout $x \in \mathbb{R}$. Cependant, numériquement, lorsque h devient très petit, le résultat obtenu par Python s'écarte de cette valeur théorique. En effet, les erreurs d'arrondi liées aux nombres flottants et la soustraction de quantités très proches (annulation numérique) peuvent empêcher la convergence vers 2.

Exercice 1.10 (Conversion j/h/m/s ⇝ s)

Les durées peuvent être exprimées en secondes ou en heures-minutes-secondes. Ainsi, la durée 0 jours 6 h 34 min et 12 s peut être exprimée par 21612 secondes en tout.

Affecter les variables `jours`, `heures`, `minutes` et `secondes`. Calculer `secondes_totales`, le nombre total de secondes correspondant.

Exemple : si `jours, heures, minutes, secondes = 1, 1, 30, 2`, on devra obtenir `secondes_totales = 91802`

Correction

```
jours, heures, minutes, secondes = 1, 1, 30, 2
secondes_totales = ((jours * 24 + heures) * 60 + minutes) * 60 + secondes
print(secondes_totales)
```

```
91802
```

Exercice 1.11 (Conversion s ⇝ j/h/m/s)

Affecter à la variable `secondes_totales` un nombre de secondes. Calculer le nombre de jours, d'heures, de minutes et de secondes correspondants. Exemple : si `secondes_totales = 125462`, on devra obtenir 1 jours 10 heures 51 minutes et 2 secondes.

Suggestion : `divmod`.

Correction

```
secondes_totales = 125462  # 1 jour, 10 heures, 51 minutes et 2 secondes
jours, reste = divmod(secondes_totales, 24*60*60)
heures, reste = divmod(reste, 60*60)
minutes, secondes = divmod(reste, 60)
# Affichage amélioré
print(f"{secondes_totales} secondes correspondent à {jours}j {heures}h {minutes}m {secondes}s")
```

Attention à ne pas utiliser le nom `min` pour les minutes, car `min` est une fonction prédéfinie en Python.[8]

Exercice Bonus 1.12 (Années martiennes — *cf.* cours N. Meloni)

Affecter à la variable `aT` un nombre d'années terrestres. Calculer le nombre de jours et d'années martiens correspondants sachant que

- 1 an terrestre = 365.25 jours terrestres
- 1 jour terrestre = 86400 secondes
- 1 jour martien = 88775.244147 secondes
- 1 an martien = 668.5991 jours martiens

On arrondira les valeurs à l'entier le plus proche en utilisant la fonction `round(x)` (*e.g.* `round(3.7)` vaut 4).

Par exemple, si `aT=36.5`, on doit obtenir "19 année(s) et 272 jour(s) martiens "

Correction

On ne peut pas utiliser `divmod` car ici on ne travaille pas avec des diviseurs entiers.

```
aT = 36.5
jT = aT*365.25
s  = jT*86400
jM = s/88775.244147
aM = round(jM/668.5991)
print(aM, "année(s) et", round(jM-(aM*668.599)), "jour(s) martiens")
# soit encore
# print(f"""{aT} an(s) terrestre(s) correspond(ent) à \
# {aM} an(s) et {round(jM-(aM*668.599))} jour(s) martiens""")
```

Exercice 1.13 (Paradoxe de Simpson : la difficulté de comparer des ratios)

Deux médicaments, A et B, ont été testés pour traiter une maladie grave. Les données recueillies concernent 26 personnes réparties par genre (hommes et femmes) et par traitement reçu (A ou B). Les données sont présentées dans les tableaux suivants :

Médicament A

Genre	Total	Guéris
Hommes	5	1
Femmes	9	4

Médicament B

Genre	Total	Guéris
Hommes	10	3
Femmes	2	1

Calculez le taux de guérison pour chaque groupe (hommes et femmes) traité avec chaque médicament. Combinez ensuite les données pour obtenir les taux de guérison globaux pour chaque médicament. Vérifiez en particulier que le médicament B a un taux de guérison supérieur au médicament A pour les hommes, le médicament B a aussi un taux de guérison supérieur au médicament A pour les femmes, mais il a un taux de guérison inférieur pour la population globalement. Il s'agit du Paradoxe de Simpson.

Généralisez l'exercice pour l'exemple suivant (issu de https://freakonometrics.hypotheses.org/231) :

8. Cette écriture `f"..."` est appelée **f-string** ; on l'utilise ici pour rendre l'affichage plus lisible mais sa syntaxe ne sera présentée en détail qu'au chapitre suivant.

Personnes Saines

Hôpital	Total	Survivants
A	600	590
B	900	870

Personnes Malades

Hôpital	Total	Survivants
A	400	210
B	100	30

Correction
Taux de guérisons de chaque médicament pour les hommes, pour les femmes et pour la totalité de la population :

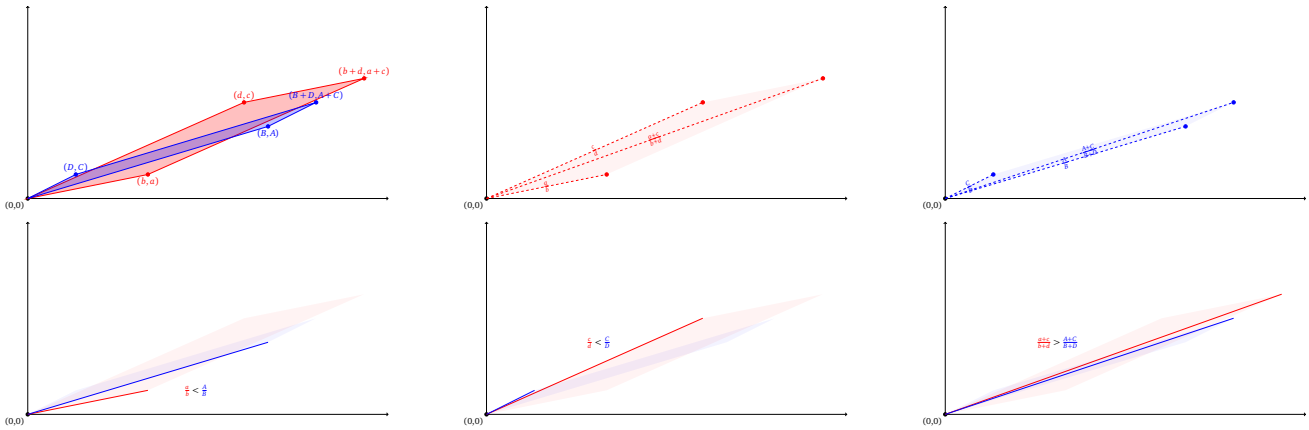
	Médicament A	Médicament B
Hommes Guéris / Hommes en tout	$1/5 = 20\%$	$3/10 = 30\%$
Femmes Guéries / Femmes en tout	$4/9 = 44\%$	$1/2 = 50\%$
Personnes Guéries / Personnes en tout	$5/14 = 36\%$	$4/12 = 33\%$

Le médicament B a un taux de guérison supérieur au médicament A pour les hommes ($20\% < 30\%$), le médicament B a aussi un taux de guérison supérieur au médicament A pour les femmes ($44\% < 50\%$), cependant il a un taux de guérison inférieur pour la population globalement ($36\% > 33\%$). Il s'agit du PARADOXE DE SIMPSON. Le paradoxe se produit car l'échantillon qu'on étudie n'est pas distribué de manière homogène : https://scienceetonnante.com/2013/04/29/le-paradoxe-de-simpson/

En effet, il est délicat d'estimer des ratios : on peut avoir les deux inégalités du système de gauche mais cela n'implique pas l'inégalité de droite :

$$\begin{cases} \frac{a}{b} < \frac{\mathrm{A}}{\mathrm{B}} \\ \frac{c}{d} < \frac{\mathrm{C}}{\mathrm{D}} \end{cases} \quad \not\Rightarrow \quad \frac{a+c}{b+d} < \frac{\mathrm{A}+\mathrm{C}}{\mathrm{B}+\mathrm{D}}.$$

Dans notre exemple $a/b = 1/5$, $\mathrm{A}/\mathrm{B} = 3/10$, $c/d = 4/9$, $\mathrm{C}/\mathrm{D} = 1/2$ et $(a+b)/(c+d) = 5/14$, $(\mathrm{A}+\mathrm{B})/(\mathrm{C}+\mathrm{D}) = 4/12$. On peut visualiser ces inégalités en comparant les pentes des coté de deux parallélogrammes et les pentes de leurs diagonales :



Cela signifie que, même si le taux de guérison est inférieur pour le médicament A par rapport au médicament B dans chaque groupe (hommes et femmes), la combinaison des groupes peut inverser cette tendance. En d'autres termes, LA TENDANCE OBSERVÉE DANS CHAQUE SOUS-GROUPE PEUT DISPARAÎTRE OU S'INVERSER LORSQUE LES GROUPES SONT COMBINÉS.

Dans notre exemple, bien que le médicament B ait des taux de guérison supérieurs dans chaque sous-groupe (hommes et femmes), le médicament A peut avoir un taux de guérison global supérieur une fois que les données des deux groupes sont combinées. Cela se produit en raison des différentes tailles de groupe et des proportions de guéris dans chaque sous-groupe, ce qui illustre bien le paradoxe de Simpson.

Voici comment obtenir ces résultats :

```
# Données
HAG, HA = 1, 5    # Hommes avec méd. A : guéris, total
HBG, HB = 3, 10   # Hommes avec méd. B : guéris, total
FAG, FA = 4, 9    # Femmes avec méd. A : guéries, total
FBG, FB = 1, 2    # Femmes avec méd. B : guéries, total

# Population totale
PAG, PA = HAG + FAG, HA + FA  # Population avec méd. A : guéries, total
PBG, PB = HBG + FBG, HB + FB  # Population avec méd. B : guéries, total
```

```python
# Taux de guérison
HAT = 100 * HAG / HA
FAT = 100 * FAG / FA
HBT = 100 * HBG / HB
FBT = 100 * FBG / FB
PAT = 100 * PAG / PA
PBT = 100 * PBG / PB

# Affichage
print("Pop.       | Médicament A             | Médicament B")
print("-----------|--------------------------|--------------------------")
print(f"Hommes     | Guéris: {HAG}/{HA} ({HAT:.1f}%)      | Guéris: {HBG}/{HB} ({HBT:.1f}%)")
print(f"Femmes     | Guéris: {FAG}/{FA} ({FAT:.1f}%)      | Guéris: {FBG}/{FB} ({FBT:.1f}%)")
print("-----------|--------------------------|--------------------------")
print(f"Total      | Guéris: {PAG}/{PA} ({PAT:.1f}%)     | Guéris: {PBG}/{PB} ({PBT:.1f}%)")
```

```
Pop.       | Médicament A             | Médicament B
-----------|--------------------------|--------------------------
Hommes     | Guéris: 1/5 (20.0%)      | Guéris: 3/10 (30.0%)
Femmes     | Guéris: 4/9 (44.4%)      | Guéris: 1/2 (50.0%)
-----------|--------------------------|--------------------------
Total      | Guéris: 5/14 (35.7%)     | Guéris: 4/12 (33.3%)
```

CHAPITRE 2

Structures de référence : Chaînes de caractères, Listes, Tuples, Dictionnaires et Ensembles

Python propose différents **structures** pour stocker des éléments :

- `str` : chaîne de n caractères **indexés de** 0 **à** $n-1$ qu'**on ne peut pas modifier** ;
- `list` : tableau de n éléments **indexés de** 0 **à** $n-1$ **modifiable** (on peut ajouter ou retirer des éléments) ;
- `tuple` : tableau de n éléments **indexés de** 0 **à** $n-1$ qu'**on ne peut pas modifier** ;
- `range` : itérateur produisant une séquence d'entiers (souvent utilisé pour itérer sur une série d'entiers dans une boucle) ;
- `dict` : tableau d'éléments **indexés par des types immuables** auquel on peut ajouter ou retirer des éléments ;
- `set` : tableau d'**éléments uniques** non indexés.

https://docs.python.org/fr/3/tutorial/datastructures.html

2.1. Les chaînes de caractères (String) et la fonction `print`

En programmation, le texte s'appelle généralement *chaîne de caractères* ou simplement *chaîne* (*string* en anglais). On peut imaginer une chaîne comme une suite de lettres, chiffres et symboles. Par exemple, le premier programme que nous avons écrit, `Hello world`, utilisait déjà une chaîne.

Créer des chaînes Pour créer une *chaîne de caractères*, il faut placer le texte entre guillemets doubles `"..."` ou simples `'...'` :[1]

```
s = "Topolino et Minnie"
s = 'Topolino et Minnie'
```

Si la chaîne contient une apostrophe, il est conseillé d'utiliser des guillemets doubles pour éviter toute ambiguïté. Par exemple, la chaîne suivante provoquerait une erreur :

```
s = 'Aujourd'hui'   # erreur : la chaîne se ferme trop tôt
```

alors que celle-ci fonctionne correctement

```
s = "Aujourd'hui"
```

Notez que Python, lorsqu'il affiche une chaîne (par exemple via la fonction `print()` ou dans l'interpréteur), la montre toujours entre guillemets simples, quel que soit le type de guillemets utilisé dans le code source.[2]

Concaténation de chaînes Assembler plusieurs chaînes s'appelle une *concaténation*. L'opérateur + sert à les assembler bout à bout :

```
s1 = "Bonjour"
s2 = " le monde"
s = s1 + s2
print(s)
```

```
Bonjour le monde
```

Attention à bien gérer les espaces : ici, c'est la deuxième chaîne qui commence par une espace.

Nota bene : l'opérateur + ne fonctionne **que** pour des chaînes de caractères. Si on essaie de concaténer une chaîne et un nombre, Python affiche une erreur :

1. Remarques pratiques : sur un clavier AZERTY, le `"` se trouve sur la touche du chiffre 3, et l'apostrophe `'` sur la touche du chiffre 4. Attention à ne pas confondre l'apostrophe `'` avec l'accent grave `` ` `` (situé sur la touche du chiffre 7), qui n'a pas la même signification en Python.

2. Il est également possible d'utiliser des triples guillemets (`'''` ou `"""`) pour créer des chaînes multilignes. Ces notations permettent d'inclure directement des retours à la ligne dans le texte. À noter que certains langages (comme C, C++ ou Java) distinguent nettement les guillemets simples et doubles : les premiers servent aux caractères isolés, les seconds aux chaînes de caractères.

```
>>> year = 2023
>>> s = "Nous sommes en " + year
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Pour résoudre cela, on doit convertir (*caster*) explicitement le nombre en chaîne à l'aide de la fonction `str()` :

```
year = 2023
s = "Nous sommes en " + str(year)
print(s)
```

```
Nous sommes en 2023
```

Différence avec les nombres Les langages de programmation distinguent les chaînes de caractères des nombres. En Python, les guillemets (simples ou doubles) déterminent que la donnée est une chaîne et non une valeur numérique :

```
>>> type('912'), type("912"), type(912)
(<class 'str'>, <class 'str'>, <class 'int'>)
```

Ainsi, `'912'` et `"912"` sont des chaînes de caractères, car elles sont entourées de guillemets ; `912` est un nombre entier (`int`). Autrement dit, même si les caractères d'une chaîne représentent des chiffres, Python ne les considère pas comme des nombres : ils ne peuvent donc pas être additionnés ou multipliés comme tels.

```
print("2" + "3")  # concaténation → "23"
print(2 + 3)      # addition → 5
```

```
23
5
```

Remarque : on vient de voir que l'opérateur "+" peut avoir différents buts en fonction des types de variables qu'on manipule :

- avec des types numériques, il sert à additionner ;
- avec des chaînes de caractères, il sert à concaténer.

Caractères spéciaux dans les chaînes Certains caractères spéciaux ne s'affichent pas directement : ils contrôlent la mise en forme du texte, comme les tabulations ou les retours à la ligne. Ils sont très utiles pour organiser les sorties de programme.

- **Tabulation (`\t`)**
  Une tabulation est un caractère spécial qui déplace le curseur jusqu'au prochain *tab stop*, c'est-à-dire la prochaine position prédéfinie sur la ligne. Elle ne correspond pas à une simple espace : elle sert à aligner du texte, par exemple pour simuler un petit tableau dans la console.

  ```
  s1 = "A\tB\tC"
  s2 = "AA\tBB\tCC"
  print(s1)
  print(s2)
  ```

  ```
  A       B       C
  AA      BB      CC
  ```

- **Retour à la ligne (`\n`)**
  Un retour à la ligne (ou retour chariot) est un caractère spécial indiquant la fin d'une ligne et le passage à la suivante. Exemple :

  ```
  s = "Ligne1\nLigne2"
  print(s)
  ```

  ```
  Ligne1
  Ligne2
  ```

  On peut combiner retours à la ligne et tabulations pour créer des tableaux lisibles :

  ```
  print("Nom\tÂge\tVille\nAlice\t25\tParis\nBob\t30\tLyon")
  ```

  ```
  Nom     Âge     Ville
  Alice   25      Paris
  Bob     30      Lyon
  ```

- **Chaînes multilignes**

  On peut aussi écrire une chaîne sur plusieurs lignes en utilisant des triples guillemets `"""..."""` ou des triples apostrophes `'''...'''`. Dans ce cas, les retours à la ligne du code source sont automatiquement conservés dans la chaîne.

```
s = """Coucou!
Je suis là."""
print(s)
```

```
Coucou!
Je suis là.
```

  Si l'on veut au contraire empêcher le passage à la ligne, on peut ajouter un `\` à la fin de chaque ligne :

```
s = """\
Coucou!\
Je suis là.\
"""
print(s)
```

```
Coucou!Je suis là.
```

La fonction `print` La fonction `print(object)` sert à afficher des objets à l'écran. Par défaut, **elle ajoute un retour à la ligne.** L'argument `object` peut être une chaîne de caractères, un nombre, etc.

```
print("Ciao")     # affiche une chaîne
print(2022)       # convertit automatiquement l'entier en chaîne
```

```
Ciao
2022
```

Pour ne pas ajouter de retour à la ligne, on peut passer l'argument optionnel `end` :

```
print("Ciao", end="")
print(2022)  # ici pas de saut de ligne entre les deux prints
```

```
Ciao2022
```

Pour afficher plusieurs objets séparés par un espace il suffit de les passer à la fonction séparés par une virgule :

```
print("J'ai", 7, "ans")  # conversion automatique en string et espace ajouté
```

```
J'ai 7 ans
```

Pour supprimer l'espace automatique entre les objets, on peut utiliser l'argument optionnel `sep` :

```
print("J'ai", 7, "ans", sep="") # supprime les espaces
```

```
J'ai7ans
```

Il est aussi courant de préparer une seule chaîne avant de la passer à `print` :

```
s = "J'ai " + str(7) + " ans" # concaténation classique
print(s)

age = 7
s = f"J'ai {age} ans"         # f-string décrites à la page 34
print(s)
```

```
J'ai 7 ans
J'ai 7 ans
```

Cette dernière approche permet de mieux contrôler l'affichage, notamment lorsqu'on combine texte et variables numériques, sans se soucier des conversions automatiques.

Opérations, fonctions et méthodes. En Python, il existe plusieurs façons d'effectuer des actions sur des données : les opérations, les fonctions et les méthodes. Voici leurs différences principales :

Opérations. Actions simples réalisées à l'aide d'opérateurs (+, -, *, etc.).

Fonctions. Blocs de code autonomes appelés sans objet spécifique (`fonction()`). Elles peuvent prendre des arguments en entrée, effectuer des opérations et retourner un résultat.

Méthodes. Similaires aux fonctions, mais attachées à un objet spécifique (`objet.methode()`). Elles peuvent interroger ou modifier l'état de l'objet.

Exemples d'opérations sur les chaînes

Expression	Description
`s1 + s2`	Concatène `s1` et `s2`.
`s1 * n`	Répète `s1` *n* fois (avec *n* entier positif).
`"abc" in s`	`True` si `"abc"` est dans `s`, `False` sinon.

Exemples de fonctions sur les chaînes

Expression	Description
`len(s)`	Renvoie le nombre de caractères de la chaîne `s`.
`str(n)`	Transforme un nombre `n` en chaîne de caractères.

```
>>> s = "="
>>> print(30*s + " ♡ " + 30*s)
============================== ♡ ==============================
>>> print("=" in s)
True
>>> print("X" in s)
False
>>> print(len(s))
1
```

Exemples de méthodes sur les chaînes

Expression	Description
`s.index("x")`	Indice de la première occurrence de `"x"` dans `s`.
`s.count("x")`	Nombre d'occurrences de `"x"` dans `s`.
`s.replace("x","y")`	Nouvelle chaîne où chaque `"x"` est remplacé par `"y"`.
`s.lower()`	Nouvelle chaîne avec tous les caractères en minuscule.
`s.upper()`	Nouvelle chaîne avec tous les caractères en majuscule.
`s.capitalize()`	Nouvelle chaîne avec la première lettre du premier mot en majuscule.
`s.title()`	Nouvelle chaîne avec la première lettre de chaque mot en majuscule.
`s.split()`	Liste de mots de la chaîne, séparés par les espaces.
`s.join(liste)`	Construit une chaîne en concaténant les éléments de `liste` séparés par `s`.
`s.isalpha()`	Vrai si tous les caractères de `s` sont alphabétiques.
`s.isdigit()`	Vrai si tous les caractères de `s` sont des chiffres.
`s.isalnum()`	Vrai si tous les caractères de `s` sont alphabétiques ou numériques.
`s.isspace()`	Vrai si tous les caractères de `s` sont des espaces (espaces, tabulations, retours ligne).
`s.startswith("x")`	Vrai si la chaîne commence par `"x"`.
`s.endswith("x")`	Vrai si la chaîne se termine par `"x"`.
`s.strip()`	Nouvelle chaîne sans les espaces (ou caractères donnés) au début et à la fin.
`s.lstrip()`	Nouvelle chaîne sans les espaces au début.
`s.rstrip()`	Nouvelle chaîne sans les espaces à la fin.
`s.find("x")`	Indice de la première occurrence de `"x"` dans `s`, ou `-1` si absente.

Découverte des méthodes disponibles Pour connaître toutes les méthodes associées à une variable (par exemple une chaîne), on peut utiliser `dir()` :

```
>>> s = "ma chaîne"
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
↪ '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getstate__', '__gt__', '__hash__',
↪ '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__',
↪ '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
↪ '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center',
↪ 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
↪ 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
↪ 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
↪ 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
↪ 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
↪ 'zfill']
```

Pour obtenir l'aide sur une méthode spécifique, on utilise `help()` :

```
>>> s = "ma chaîne"
>>> help(s.split) # ne pas mettre les parenthèses à la suite du nom de la méthode
Help on built-in function split:

split(sep=None, maxsplit=-1) method of builtins.str instance
    Return a list of the substrings in the string, using sep as the separator string.

      sep
        The separator used to split the string.

        When set to None (the default value), will split on any whitespace
        character (including \n \r \t \f and spaces) and will discard
        empty strings from the result.
      maxsplit
        Maximum number of splits.
        -1 (the default value) means no limit.

    Splitting starts at the front of the string and works to the end.

    Note, str.split() is mainly useful for data that has been intentionally
    delimited.  With natural text that includes punctuation, consider using
    the regular expression module.
```

Exemples d'utilisation combinée

```
>>> string1 = "Minnie"
>>> string2 = "Topolino"

>>> # Concaténation et remplacement
>>> s3 = string1 + " et " + string2
>>> s4 = s3.replace("et","&")
>>> print(s3)
Minnie et Topolino
>>> print(s4)
Minnie & Topolino

>>> # Remplacement multiple dans une chaîne
>>> texte = "Une fonctttion ttttrès prattttique si vous répétttttez ttttrop les tttt"
>>> print(texte.replace("tttt","t"))
Une fonction très pratique si vous répétez trop les t

>>> # Split et join
>>> s5 = s4.split(" & ")
>>> print(s5)
['Minnie', 'Topolino']
>>> s6 = " ou ".join(s5)
```

```
>>> print(s6)
Minnie ou Topolino

>>> # Centre avec remplissage
>>> print(s6.center(30,"-"))
------Minnie ou Topolino------

>>> # Recherche et comptage
>>> print(s6.index("n"))
2
>>> print(s6.count("n"))
3

>>> chaine = "Buon compleanno a te, buon compleanno a te, \
... buon compleanno caro Pluto, buon compleanno a te!"
>>> print(chaine.count("buon"))
3

>>> # Split sur espace et join
>>> s = "Pablo Neruda Saint martin d'hères"
>>> l = s.split()
>>> print(l)
['Pablo', 'Neruda', 'Saint', 'martin', "d'hères"]
>>> print(' - '.join(l))
Pablo - Neruda - Saint - martin - d'hères

>>> # Minuscule, majuscule, capitalisation
>>> print("Pablo Neruda".lower(), "Pablo Neruda".upper(), "pablo neruda".capitalize(), "pablo
↪  neruda".title())
pablo neruda PABLO NERUDA Pablo neruda Pablo Neruda

>>> # Split avec séparateur personnalisé
>>> s = "J. M. Brown AND B. Mencken AND R. P. van't Rooden"
>>> print(s.split(' AND '))
['J. M. Brown', 'B. Mencken', "R. P. van't Rooden"]
```

La fonction `input` La fonction `input()` sert à demander une saisie à l'utilisateur. Elle prend en argument un message (une chaîne de caractères), affiche ce message à l'utilisateur, puis renvoie la donnée entrée **sous forme de chaîne de caractères**.

Exemple :

```
x = input("Choisis un nombre : ")
print(f"Tu as choisi {x}")
```

Sortie possible :

```
Choisis un nombre : 3
Tu as choisi 3
```

Si la donnée est censée être un nombre, il faut la convertir avec `int()` ou `float()` avant tout calcul. On peut aussi utiliser `eval(x)` pour évaluer directement l'expression, mais `eval()` doit être utilisé avec prudence pour des raisons de sécurité.
NB : nous n'utiliserons jamais `input()` dans ce cours.

2.1.1. `f-string` : interpolation de chaînes de caractères et écriture formatée

L'interpolation de chaînes de caractères consiste à insérer automatiquement, à l'intérieur d'une chaîne, la valeur de variables ou le résultat d'expressions. En Python, on peut utiliser les `f-string`, introduites à partir de la version 3.6.

Syntaxe des f-strings Dans une `f-string`, le préfixe `f` placé devant la chaîne indique à Python qu'il doit évaluer les expressions écrites entre accolades `{}` et remplacer ces espaces réservés par leurs valeurs. Pour créer une f-string :

1. écrire la chaîne de caractères avec les noms des variables que l'on souhaite afficher ;
2. ajouter le préfixe `f` devant la chaîne ;
3. entourer chaque variable d'accolades `{}`. Python remplacera chaque nom par sa valeur.

Dans l'exemple suivant, les variables `name` et `program` sont remplacées par leur contenu respectif lors de l'affichage.

```
name = 'World'
program = 'Python'
print(f"Hello {name}! This is {program}.")
```

```
Hello World! This is Python.
```

Expressions dans les f-strings Les f-strings peuvent contenir des expressions Python arbitraires, y compris de l'arithmétique :

```
a, b = 12, 3
print(f'{a} x {b} = {a*b}')
```

```
12 x 3 = 36
```

Différence avec `print` classique La fonction `print` convertit automatiquement les objets en chaînes et ajoute des espaces entre eux :

```
n1, n2 = 9, 10
print("Sara a", n1, "ans et Lorenzo", n2, ".") # espace automatique indésirable
print("Sara a ", n1, " ans et Lorenzo ", n2, ".", sep="") # contrôle complet
```

```
Sara a 9 ans et Lorenzo 10 .
Sara a 9 ans et Lorenzo 10.
```

Remarque 5 (Self-documenting expressions)
Depuis Python 3.8, on peut écrire directement `f'{variable = }'` au lieu de `f'variable = {variable}'` pour afficher à la fois le nom et la valeur de la variable :

```
>>> a = 3
>>> print(f'{a = }')
a = 3
```

Formatage numérique dans les f-strings Les f-strings permettent aussi de contrôler le format des nombres, leur alignement et le nombre de décimales :

```
>>> a, n = -1234.56789, 9876

>>> print(f'a = {a}, n = {n}')
a = -1234.56789, n = 9876

>>> print(f'a = {a:g}, n = {n:g}')    # format le plus approprié
a = -1234.57, n = 9876
>>> print(f'{a:.3e}')                 # notation scientifique
-1.235e+03
>>> print(f'{a:.2f}')                 # 2 décimales
-1234.57
>>> print(f'{a:12.2f}')               # largeur totale 12, 2 décimales
    -1234.57

>>> print(f'{123:{0}{6}}')            # 6 chiffres, complété par des 0
000123

>>> print(f'{a:>12.2f}')              # justifié à droite
    -1234.57
>>> print(f'{a:<12.2f}')              # justifié à gauche
-1234.57
>>> print(f'{a:^12.2f}')              # centré
  -1234.57
>>> print(f'{a:+.2f}')                # affiche toujours le signe
-1234.57
```

Remarque 6 (Remplissage dans les f-strings)
Python permet de contrôler l'alignement et le remplissage des chaînes dans les f-strings. Exemples :

```
>>> word = "Ciao"
>>> print(f"{word:=<20}")  # rempli avec '=' à gauche, texte aligné à gauche
Ciao================
>>> print(f"{word:->20}")  # rempli avec '-' à gauche, texte aligné à droite
----------------Ciao
>>> print(f"{word:/^20}")  # rempli avec '/' autour, texte centré
////////Ciao////////
```

Remarque 7 (*stringprefix*)
Un *stringprefix* modifie la manière dont Python interprète une chaîne de caractères. Il doit être collé directement à la chaîne, sans espace.

Les principaux *stringprefix* sont :

- **f** pour *formatted string* : permet l'écriture formatée et l'interpolation de variables.

```
animal = "renard"
animal2 = "poulain"

# f-string : interpolation active
s = f"Le {animal} est un animal gentil.\nLe {animal2} aussi."
print(s)

# chaîne normale : pas d'interpolation
s = "Le {animal} est un animal gentil.\nLe {animal2} aussi."
print(s)
```

```
Le renard est un animal gentil.
Le poulain aussi.
Le {animal} est un animal gentil.
Le {animal2} aussi.
```

- **r** pour *raw string* : force Python à ne pas interpréter les caractères spéciaux comme `\n` ou `\t`.

```
# chaîne normale : \n crée un retour à la ligne
s = "Voici un retour à la ligne\nEt là une autre ligne"
print(s)

# raw string : \n affiché littéralement
s = r"Voici un retour à la ligne\nEt là une autre ligne"
print(s)
```

```
Voici un retour à la ligne
Et là une autre ligne
Voici un retour à la ligne\nEt là une autre ligne
```

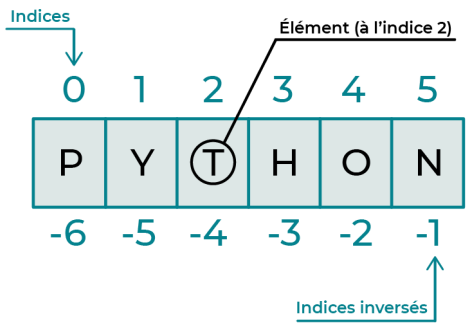
L'ajout du préfixe "r" fait que Python traite le `\n` comme deux caractères distincts : un *backslash* et un "n". L'affichage montre souvent deux *backslashes* pour indiquer un *backslash* littéral.

2.1.2. Accès et modification de chaînes de caractères

Les éléments d'une chaîne sont *indexés à partir de* 0 (et non de 1).

Accéder à un seul caractère On peut extraire un caractère par son indice : `s[i]` renvoie le $(i+1)^{e}$ caractère de la chaîne `s`.

Pour la chaîne `"PYTHON"`, l'indice "2" ou l'indice inversé "-4" correspond au troisième caractère "T" :




```
>>> s = "PYTHON"
>>> print(s[2])
T
>>> print(s[-4])
T
```

1. **Index hors limite** Si l'on utilise un indice supérieur ou égal à la longueur de la chaîne, Python renvoie une erreur :

```
>>> s = "Topolino"
>>> print(s[8])  # IndexError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

2. **Immuabilité des chaînes** Une chaîne de caractères est un objet **immuable**: sa longueur est fixe et ses caractères ne peuvent pas être modifiés individuellement. Tenter d'affecter un caractère existant provoque une erreur:

```
>>> s    = "Press return to exit"
>>> s[0] = "p"  # TypeError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Pour modifier une chaîne, il faut **redéfinir entièrement la chaîne**, par exemple en utilisant la concaténation et le *slicing* `s = "p" + s[1:]` (voir ci-dessous).

Extraction d'une sous-chaîne (*slicing*) Soit `s` une chaîne de caractères. On peut extraire une **sous-chaîne** en utilisant la notation par tranches (*slicing*).

- **Début et fin**: indices i (inclus) et j (exclu) séparés par deux-points

  `s[i:j]` équivaut à `s[i]+s[i+1]+s[i+2]+...+s[j-1]`

  Les dépassements d'indices sont autorisés: si j est supérieur à la longueur de la chaîne, Python prend jusqu'à la fin.
- **Début, fin et pas**: indices i (début), j (fin) et k (pas)

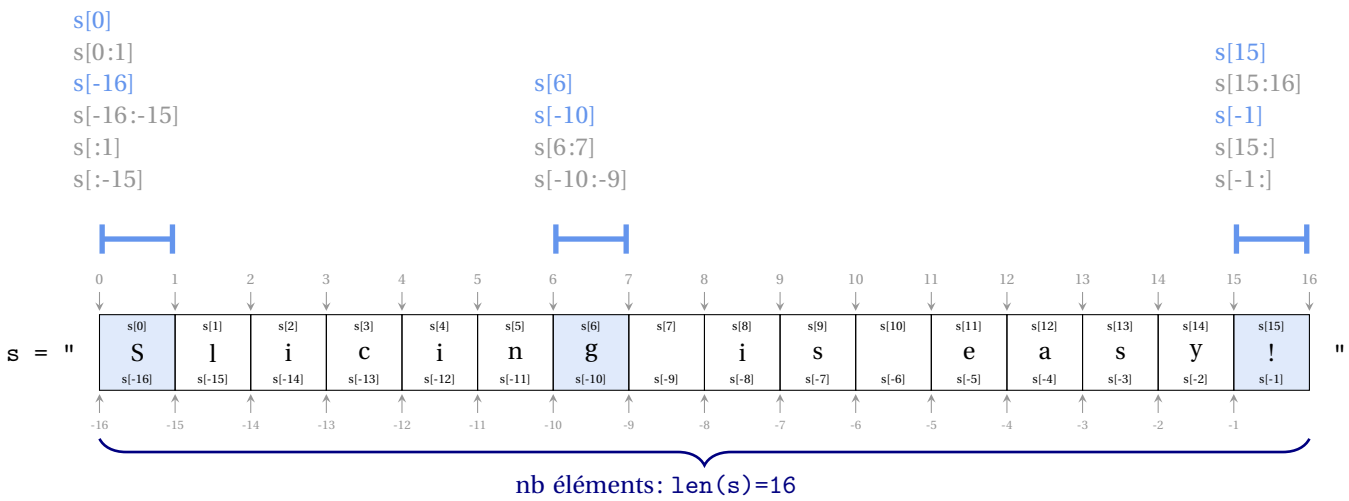
  `s[i:j:k]` équivaut à `s[i]+s[i+k]+s[i+2k]+\dots+s[i+mk]` avec `i+mk<j`

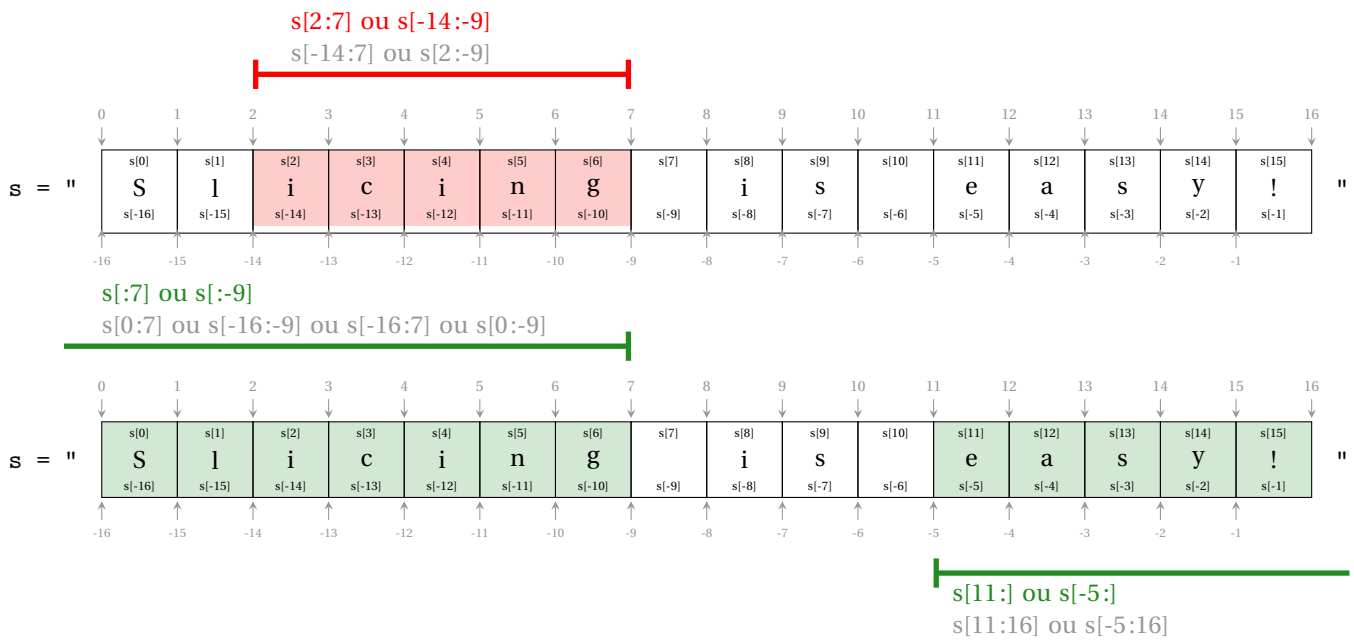
  Le pas k peut être négatif, auquel cas $j < i$ pour parcourir la chaîne en sens inverse. Sans indication, le pas k est égale à 1.

Exemples de slices pour une chaîne de longueur 9 (indices de 0 à 8).

Slice	Indices sélectionnés
`[2:7]`	2, 3, 4, 5, 6
`[2:7:2]` ou `[2:8:2]`	2, 4, 6
`[2:]`	2, 3, 4, 5, 6, 7, 8 (jusqu'à la fin)
`[:4]`	0, 1, 2, 3 (du début jusqu'à 3)
`[2::3]`	2, 5, 8 (jusqu'à la fin avec un pas de 3)
`[:6:2]`	0, 2, 4 (du début jusqu'à 5 avec un pas de 2)
`[7:5:-1]`	7, 6
`[7:2:-2]`	7, 5, 3
`[7::-1]`	7, 6, 5, 4, 3, 2, 1, 0
`[:3:-1]`	8, 7, 6, 5, 4
`[:]`	0, 1, 2, 3, 4, 5, 6, 7, 8 (tous)
`[::-1]`	8, 7, 6, 5, 4, 3, 2, 1, 0 (tous en sens inverse)

Des petits dessins permettront de bien comprendre cette opération:

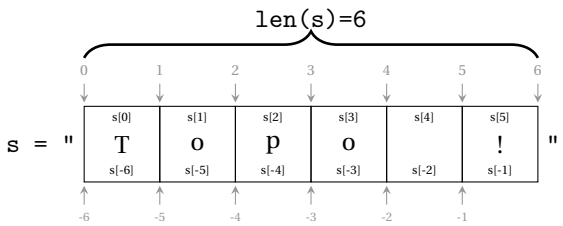




Attention aux instructions suivantes qui renvoient une chaîne vide :

```
>>> s = "Slicing is easy!"
>>> print( s[-5:0], s[11:0], s[-1:0] )
```

Un autre exemple avec indication explicite du pas.



Nous avons une chaîne de 6 éléments dont les indices vont de 0 à 5. Si on utilise un indice supérieur à 5 ou inférieur à −6 on obtient une erreur. Cependant, avec le *slicing*, le dépassement des indices est permis :

```
>>> s = "Topo !"

>>> print(s[::])
Topo !
>>> print(s[::-1])
! opoT

>>> print(s[2:9])  # dépassement à droite (>len(s))
po !
>>> print(s[-4:9]) # dépassement à droite (>len(s))
po !

>>> print(s[-8:-3]) # dépassement à gauche (<-len(s))
Top
>>> print(s[-8:3])  # dépassement à gauche (<-len(s))
Top
```

Remarque 8 (Pourquoi le premier indice est inclus et le dernier exclu ?)
Chaque élément d'une liste ou d'une chaîne est associé à un indice, mais on peut aussi le considérer comme situé entre deux positions et donc associé à deux indices, comme le montrent les flèches numérotées dans les exemples précédents. Dans la notation `s[2:7]`, les éléments sélectionnés sont ceux compris entre les flèches 2 et 7, c'est-à-dire les indices 2 à 6 inclus.

Cette représentation (éléments situés entre deux flèches) rend plus naturel le calcul de la longueur d'une sous-chaîne ou d'une tranche (`len(s[i:j]) = j-i`).

Attention Il existe deux façons de comprendre l'extraction de sous-chaînes (*slicing*) :

1. considérer les indices des éléments, en prenant tous les éléments dont les indices se situent entre le début inclus et la fin exclue ;
2. considérer les bornes comme délimitant les éléments, ce qui est utile pour visualiser les tranches (cependant, il faut être prudent lorsque le pas est négatif).

Exemples avec la chaîne `s = "Topo !"` :

```
>>> print(s[-1:-5:1])   # vide, car le pas est positif mais l'intervalle est inversé

>>> print(s[-1:-5:-1])  # s[-1], s[-2], s[-3], s[-4]
! op
>>> print(s[5:1:-1])    # s[5], s[4], s[3], s[2]
! op

>>> print(s[:-5:-1])    # s[-1], s[-2], s[-3], s[-4]
! op
>>> print(s[:1:-1])     # s[5], s[4], s[3], s[2]
! op

>>> print(s[-1::-1])    # s[-1], s[-2], s[-3], s[-4], s[-5], s[-6]
! opoT
>>> print(s[5::-1])     # s[5], s[4], s[3], s[2], s[1], s[0]
! opoT
```

Remarque : l'élément correspondant au **premier indice est inclus**, le **dernier est exclu**. Pour les pas négatifs, il est plus simple de raisonner en termes d'éléments à inclure plutôt qu'en termes de tranches.

2.2. Listes

Une liste en Python est une collection ordonnée d'objets, séparés par des virgules et encadrés par des crochets `[]`. Elle peut contenir des doublons, des objets de types différents, et même d'autres listes.

```
L = [1,2,3,2]
print(L)

[1, 2, 3, 2]
```

```
L = ["a","d","m"]
print(L)

['a', 'd', 'm']
```

```
L = [1,'ok',[5,10]]
print(L)

[1, 'ok', [5, 10]]
```

Pour déclarer une liste vide (à remplir ensuite) :

```
L = []        # liste vide
L = list()    # autre notation
```

Indexation des listes Les listes sont **indexées à partir de** 0.

- L'indice du premier élément est 0, le deuxième est 1, etc.
- Les indices négatifs comptent depuis la fin de la liste : −1 correspond au dernier élément, −2 à l'avant-dernier, etc.

Par exemple, pour la liste "L = [12, 11, 18, 7, 15, 3] ", l'entier 18 est associé à l'indice 2 ou à l'indice négatif -4 :

L=

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]
12	11	18	7	15	3
L[-6]	L[-5]	L[-4]	L[-3]	L[-2]	L[-1]

len(L)=6

```
L = [12, 11, 18, 7, 15, 3]
print(L[2], L[-4])

18 18
```

Découpage de listes (slicing) On peut extraire une sous-liste avec la notation `Liste[i:j]` (éléments de l'indice `i` inclus à `j` exclus) ou `Liste[i:j:k]` (avec un pas `k`).

```
>>> L = [12, 11, 18, 7, 15, 3]
>>> L[2:4]
[18, 7]
>>> L[2:]
[18, 7, 15, 3]
>>> L[:2]
[12, 11]
>>> L[:]
[12, 11, 18, 7, 15, 3]
>>> L[2:5]
[18, 7, 15]
>>> L[2:6]
[18, 7, 15, 3]
>>> L[2:7]
[18, 7, 15, 3]
>>> L[2:6:2]
[18, 15]
>>> L[-2:-4]
[]
>>> L[-4:-2]
[18, 7]
>>> L[-1]
3
```

Attention, si l'on tente d'extraire un élément en dehors de la taille de la liste, Python renvoie une erreur, mais lors de l'utilisation de la découpe, les dépassements d'indices sont autorisés.

```
>>> print( L[0], L[1], L[2], L[3], L[4], L[5] )
12 11 18 7 15 3
>>> print(L[6])   # erreur : hors limites
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> print(L[6:7]) # renvoie []
[]
```

Modification d'une liste Contrairement aux chaînes de caractères, les éléments d'une liste sont **modifiables**. On peut donc changer directement un élément par affectation à son indice :

```
L = [12, 10, 18, 7, 15, 3]
L[1] = 11       # on remplace l'élément d'indice 1 par 11
print(L)        # affiche [12, 11, 18, 7, 15, 3]
```

```
[12, 11, 18, 7, 15, 3]
```

On peut également modifier plusieurs éléments en une seule opération en utilisant le *slicing* :

```
L[2:4] = [99, 100]  # change les éléments d'indices 2 et 3
print(L)
```

```
[12, 11, 99, 100, 15, 3]
```

2.2.1. Fonctions et méthodes sur les listes

Les listes Python offrent de nombreuses opérations, fonctions et méthodes pour faciliter leur manipulation.

Opérations sur les listes Le tableau suivant présente quelques opérations usuelles applicables aux listes en Python. Ces opérations ne modifient pas les listes existantes et produisent soit une valeur booléenne, soit une nouvelle liste.

Expression	Description	Effet
`x in L`	Renvoie `True` si la liste `L` contient l'élément `x`, sinon `False`.	Renvoie
`L1 + L2`	Concatène les listes `L1` et `L2` en créant une nouvelle liste.	Renvoie
`L * n`	Crée une nouvelle liste en répétant `L` `n` fois.	Renvoie

```
L = [1, 2, 3]
print( 2 in L, 5 in L )
print( 3*L )
print( L + [4, 5] )
```

```
True False
[1, 2, 3, 1, 2, 3, 1, 2, 3]
[1, 2, 3, 4, 5]
```

Attention Certaines opérations sur les listes peuvent produire des résultats inattendus si on ne fait pas attention aux détails :

- L'opération `L * n` ne crée pas de copies indépendantes des objets contenus dans `L`. Si `L` contient des objets mutables (comme d'autres listes), toutes les occurrences pointent vers les mêmes objets.
- L'opération `L1 + L2` crée une nouvelle liste sans modifier `L1` ni `L2`. Pour ajouter des éléments à une liste existante, il est généralement plus efficace d'utiliser `append` ou `extend` que `L = L + [x, y]`.
- L'opérateur augmenté `L += [x]` modifie directement la liste existante, contrairement à `L = L + [x]` qui crée une nouvelle liste.

Fonctions sur les listes Le tableau suivant présente quelques fonctions intégrées de Python permettant de manipuler ou d'extraire des informations à partir de listes. Contrairement à la plupart des méthodes, ces fonctions ne modifient pas la liste, sauf indication contraire.

Expression	Description	Effet
`sum(L)`	Calcule la somme des éléments numériques de la liste `L`.	Renvoie
`len(L)`	Renvoie le nombre d'éléments contenus dans `L`.	Renvoie
`sorted(L)`	Renvoie une nouvelle liste contenant les éléments de `L` triés (ordre croissant par défaut). L'argument `reverse=True` permet d'inverser l'ordre.	Renvoie
`max(L)`	Renvoie le plus grand élément de la liste `L`.	Renvoie
`min(L)`	Renvoie le plus petit élément de la liste `L`.	Renvoie
`del L[i]`	**Instruction** supprimant l'élément d'indice `i` de la liste `L`.	Modifie

Attention Certaines fonctions sur les listes supposent que la liste n'est pas vide ou que ses éléments sont du bon type.

- Les fonctions `max` et `min` génèrent une erreur si elles sont appliquées à une liste vide. Pour éviter cela, elles acceptent un paramètre optionnel `default` qui sera renvoyé si la liste est vide :
```
L = []
print(max(L, default=100))  # 100
```
```
100
```
- La fonction `sum` nécessite que les éléments de la liste soient numériques.

```
L = [2, 37, 20, 83, -79, 21]
print(sum(L), len(L), max(L))
```
```
84 6 83
```
```
del L[1]
print(L)
```
```
[2, 20, 83, -79, 21]
```
```
L = [1, 2, 3]
L[0] = 21
print(L)
```
```
[21, 2, 3]
```
```
L[2:4] = [-2, -5, -1978]
print(L)
```
```
[21, 2, -2, -5, -1978]
```
```
L = [1, 3, 2, 5]
S = sorted(L)
R = sorted(L,reverse=True)
print(S,R)
```
```
[1, 2, 3, 5] [5, 3, 2, 1]
```

Méthodes sur les listes Le tableau suivant présente quelques méthodes couramment utilisées pour manipuler des listes en Python. Sauf indication contraire, ces méthodes **modifient** la liste sur laquelle elles sont appelées et ne créent pas de nouvelle liste.

Expression	Description	Effet
`L.append(x)`	Ajoute l'élément `x` à la fin de la liste `L`.	Modifie
`L.extend(M)`	Ajoute successivement à `L` tous les éléments de la liste `M`.	Modifie
`L.insert(i, x)`	Insère l'élément `x` à l'indice `i`.	Modifie
`L.remove(x)`	Supprime la première occurrence de l'élément `x`.	Modifie
`L.pop()`	Supprime et renvoie le dernier élément de la liste.	Modifie + renvoie
`L.pop(i)`	Supprime et renvoie l'élément d'indice `i`.	Modifie + renvoie
`L.index(x)`	Renvoie l'indice de la première occurrence de `x`.	Renvoie
`L.clear()`	Supprime tous les éléments de la liste `L`.	Modifie
`L.count(x)`	Renvoie le nombre d'occurrences de `x`.	Renvoie
`L.sort()`	Trie la liste `L` sur place (ordre croissant par défaut). L'argument `reverse=True` inverse l'ordre.	Modifie
`L.reverse()`	Inverse l'ordre des éléments de la liste `L` sur place.	Modifie

Attention Certaines méthodes (`remove`, `pop`, `index`) supposent que l'élément recherché est présent dans la liste : dans le cas contraire, une erreur est levée.

```
>>> a = [2, 37, 20, 83, -79, 21]
>>> print(a)
[2, 37, 20, 83, -79, 21]

>>> a.append(100)
>>> print(a)
[2, 37, 20, 83, -79, 21, 100]

>>> L = [17, 34, 21]
>>> a.extend(L)
>>> print(a)
[2, 37, 20, 83, -79, 21, 100, 17, 34, 21]

>>> a.count(21)
2
>>> a.remove(21)
>>> a.count(21)
1
>>> print(a)
[2, 37, 20, 83, -79, 100, 17, 34, 21]
```

```
>>> a.pop(4)
-79
>>> print(a)
[2, 37, 20, 83, 100, 17, 34, 21]

>>> idx = a.index(100)
>>> print(idx)
4

>>> a.reverse()
>>> print(a)
[21, 34, 17, 100, 83, 20, 37, 2]

>>> a.sort()
>>> print(a)
[2, 17, 20, 21, 34, 37, 83, 100]

>>> a.insert(2,7)
>>> print(a)
[2, 17, 7, 20, 21, 34, 37, 83, 100]
```

2.2.2. Copie d'une liste

Si `a` est une liste, la commande `b = a` ne crée pas un nouvel objet `b` mais établit simplement une référence (pointeur) vers `a`. Par conséquent, tout changement effectué sur `b` sera également répercuté sur `a` aussi et vice versa ! Nous pouvons observer cela dans les deux exemples suivants :

Exemple 1

```
>>> a = [1, 2, 3]
>>> b = a
>>> print(a == b)
True
>>> print(a is b)
True
>>> a[0] = 5
>>> print(a)
[5, 2, 3]
>>> print(b) # Pb
[5, 2, 3]
```

Exemple 2

```
>>> a = [1, 2, 3]
>>> b = a
>>> print(a == b)
True
>>> print(a is b)
True
>>> b[0] = 5
>>> print(a) # Pb!!!
[5, 2, 3]
>>> print(b)
[5, 2, 3]
```

Qu'est-ce qui se passe lorsque on copie une liste `a` avec la commande `b = a` ? En fait, une liste fonctionne comme un carnet d'adresses qui contient les emplacements en mémoire des différents éléments de la liste. Lorsque l'on écrit `b = a`, cela signifie que `b` contient les mêmes adresses que `a` (on dit que les deux listes «pointent» vers le même objet). Par conséquent, toute modification de l'objet sera visible depuis les deux alias.

Une première solution pour effectuer une copie peut être d'utiliser le *slicing*. En effet, l'opération `[:]` renvoie une nouvelle liste, ce qui résout le problème des pointeurs vers la même zone mémoire, comme le montrent l'exemple 3 suivant. Cette copie peut paraître satisfaisante... et elle l'est, mais seulement si l'on manipule des listes de premier niveau ne comportant aucune sous-liste, ce qui n'est pas le cas dans l'exemple 4.

Exemple 3

```
>>> a = [1, 2, 3]
>>> c = a[:] # idem que c = list(a)
>>> print(a == c)
True
>>> print(a is c)
False
>>> c[0] = 5
>>> print(a)
[1, 2, 3]
>>> print(c)
[5, 2, 3]
```

Exemple 4

```
>>> a = [1, 2, 3, [4,5,6]]
>>> c = a[:] # idem que c = list(a)
>>> print(a == c)
True
>>> print(a is c)
False
>>> print(a[3] is c[3]) # Pb !!
True
>>> c[3][0] = 20
>>> print(a)
[1, 2, 3, [20, 5, 6]]
>>> print(c)
[1, 2, 3, [20, 5, 6]]
```

En effet, le slicing n'effectue pas de copie récursive. En cas de sous-liste, on se retrouve face au même problème de références mémoire partagées. La solution est d'utiliser un module spécifique, le module `copy`, et sa fonction `deepcopy`, qui permet d'effectuer une copie récursive. Bien que nous n'ayons pas encore introduit l'importation de modules, voici comment créer une copie `c` de la liste `a` qui soit véritablement indépendante :

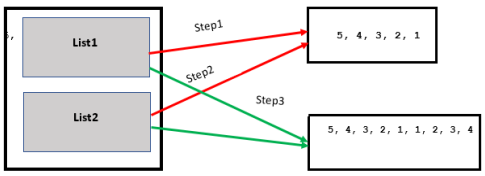
Exemple 5

```
>>> import copy
>>> a = [1, 2, 3, [4,5,6]]
>>> c = copy.deepcopy(a)
>>> print(a == c)
True
>>> print(a is c)
False
>>> print(a[3] is c[3])
False
>>> c[3][0] = 20
>>> print(a)
[1, 2, 3, [4, 5, 6]]
>>> print(c)
[1, 2, 3, [20, 5, 6]]
```

Attention (`a = a+b` ou `a = a*b` VS `a += b` ou `a *= b`) Je déconseille fortement l'utilisation des opérateurs augmentés `+=` et `*=` avec des listes, à moins que vous ne compreniez parfaitement leur fonctionnement. Voici un comportement surprenant au premier abord :

```
list1 = [5, 4, 3, 2, 1]
list2 = list1
list1 += [1, 2, 3, 4]
print(list1, id(list1))
print(list2, id(list2))
```

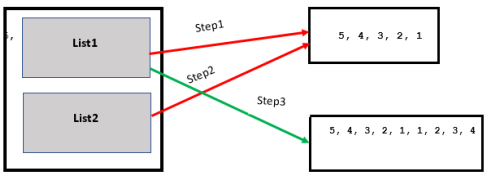
```
[5, 4, 3, 2, 1, 1, 2, 3, 4] 123202976769600
[5, 4, 3, 2, 1, 1, 2, 3, 4] 123202976769600
```



L'expression `list1 += [1, 2, 3, 4]` modifie la liste originale. Comme `list1` et `list2` pointent vers la même référence, la liste `list2` est également modifiée.

```
list1 = [5, 4, 3, 2, 1]
list2 = list1
list1 = list1 + [1, 2, 3, 4]
print(list1, id(list1))
print(list2, id(list2))
```

```
[5, 4, 3, 2, 1, 1, 2, 3, 4] 123202975623040
[5, 4, 3, 2, 1] 123202975622976
```



L'expression `list1 = list1 + [1, 2, 3, 4]` crée une nouvelle liste. Après cette instruction, `list1` a une nouvelle adresse mémoire, tandis que `list2` pointe toujours vers l'ancienne référence et n'est pas modifiée.

Plus surprenant encore: lorsqu'on tente de concaténer une liste vide avec une chaîne de caractères en utilisant l'opérateur "+", Python génère une erreur. Cela est dû au fait que l'opérateur "+" entre une liste et une chaîne de caractères n'est pas défini, provoquant une incompatibilité de type et donc une erreur:

```
>>> [] + "cochon"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

Cependant, lorsque nous utilisons l'opérateur "+=" avec une liste et une chaîne de caractères, Python considère que chaque caractère de la chaîne est ajouté comme un élément distinct à la liste. Il effectue donc une opération d'*extension* de liste:

```
>>> a = []
>>> a += "cochon"
>>> a
['c', 'o', 'c', 'h', 'o', 'n']
```

Remarque 9 (Liste récursive (auto-référente))
Les listes sont des objets mutables et les opérations comme `append` ajoutent des *références* vers des objets, et non des copies. Ainsi, avec le code

```
L = []
L.append(L)
```

la liste `L` contient une référence vers elle-même. On obtient une liste auto-référente, vérifiant `L = [L]`:

```
print( len(L) )
print( L[0] )
print( L[0] is L)
print( L == [L] )
```

```
1
[[...]]
True
True
```

Cette situation est possible car `append` ajoute l'objet passé en argument tel quel, sans le dupliquer. La liste `L` contient alors un unique élément, qui est la liste elle-même. Lors de l'affichage, Python écrit `[[...]]` afin d'éviter une récursion infinie.

2.3. Les tuples

Les tuples, ou n-uplets, sont des collections ordonnées d'objets, délimitées par des parenthèses et séparées par des virgules. Ils sont comme des listes, mais **immuables**, ce qui signifie qu'une fois créés, ils ne peuvent pas être modifiés. La distinction entre tuples et listes, avec l'immutabilité des premiers, apporte une certaine robustesse et sécurité aux données qui ne doivent pas être altérées une fois définies. L'utilisation de tuples se révèle avantageuse lorsque vous souhaitez des données qui ne doivent pas être altérées ou manipulées accidentellement. Par exemple, lors de la définition de coordonnées fixes dans un plan cartésien ou pour stocker des informations qui ne nécessitent pas de modifications ultérieures.

```
>>> T = ( 2 , 3 , ('a',5) , [4,'toto'] )
>>> print(T)
(2, 3, ('a', 5), [4, 'toto'])
```

```
>>> T = ( ) # tuple vide, idem que
>>> # T = tuple()
>>> print(T)
()
```

Si on essaye de modifier les éléments d'un tuple (qui sont des objets **immuables**, comme les chaînes de caractères) on a un message d'erreur:

```
>>> T = ( 2 , 3 , ('a',5) , [4,'toto'] )
>>> T[1] = 11
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

ATTENTION (TUPLE AVEC UN SEUL ÉLÉMENT) Afin d'éviter toute ambiguïté, **pour créer un tuple contenant un unique élément, il doit être écrit sous la forme** `(élément,)`. En l'absence de la virgule, Python interprète cela comme un parenthésage superflu, et ainsi ne crée pas un tuple.

```
>>> T = (1)
>>> print(type(T))
<class 'int'>
```

```
>>> T = (1,)
>>> print(type(T))
<class 'tuple'>
```

Opérations, fonctions et méthodes

Toute opération, fonction ou méthode qui s'applique à une liste **sans** la modifier peut être utilisée pour un tuple :

```
>>> a = (1, -10, 1, 7, "toto", 1.5)
>>> print(a)
(1, -10, 1, 7, 'toto', 1.5)
>>> print(a[2])
1
>>> a.count(1)
2
>>> a.index("toto")
4
>>> print(len(a))
6
>>> print(a+a)
(1, -10, 1, 7, 'toto', 1.5, 1, -10, 1, 7, 'toto', 1.5)
>>> print(3*a)
(1, -10, 1, 7, 'toto', 1.5, 1, -10, 1, 7, 'toto', 1.5, 1, -10, 1, 7, 'toto', 1.5)
>>> print(a+(4,5))
(1, -10, 1, 7, 'toto', 1.5, 4, 5)
```

2.4. L'itérateur `range`

La fonction `range` génère un itérateur. Plutôt que de créer et stocker une liste complète d'entiers, cette fonction produit les nombres au fur et à mesure de leur utilisation :

- `range(n)` génère un itérateur pour parcourir $[\![0;n-1]\!] = 0,1,2,\ldots,n-1$;
- `range(n,m)` génère un itérateur pour parcourir $[\![n;m-1]\!] = n,n+1,n+2,\ldots,m-1$;
- `range(n,m,p)` génère un itérateur pour parcourir $n,n+p,n+2p,\ldots,m-1$.

Attention, si on essaie d'afficher le résultat, Python ne génère pas les valeurs mais seulement l'itérateur lui-même :

```
>>> A = range(0,10)
>>> print(A)
range(0, 10)
```

Pour forcer la génération des éléments, on peut créer une liste à partir de l'itérateur avec la fonction `list` :

```
>>> A = range(0,10)
>>> A = list(A)
>>> print(A)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Voici un résumé

Opération	Description
`range(stop)`	Génère une séquence d'entiers de 0 à `stop-1`
`range(start, stop)`	Génère une séquence d'entiers de `start` à `stop-1`
`range(start, stop, step)`	Génère une séquence d'entiers en commençant par `start`, en incrémentant de `step`, et s'arrête avant le nombre `stop`.
`range(5, -1, -1)`	range inversé
`reversed(range(5))`	range inversé en utilisant la fonction `reversed()`
`range(-1, -11, -1)`	range négatif de -1 à -10
`list(range(2, 10, 2))`	Convertit `range()` en liste
`range(start, stop+step, step)`	Génère un range inclusif
`range(0, 10)[5]`	Accède directement au cinquième nombre d'un `range()`
`range(10)[3:8]`	Coupe un range pour accéder aux nombres de l'index 3 à 8
`range.start`	Obtient la valeur de départ d'un `range()`
`range.stop`	Obtient la valeur d'arrêt d'un `range()`
`range.step`	Obtient la valeur d'incrément d'un `range()`

Voici quelques exemples :

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0))
[]
>>> list(range(1))
[0]
>>> list(range(3,7))
[3, 4, 5, 6]
```

```
>>> list(range(0,20,5))
[0, 5, 10, 15]
>>> list(range(0,20,-5))
[]
>>> list(range(0,-20,-5))
[0, -5, -10, -15]
>>> list(range(20,0,-5))
[20, 15, 10, 5]
```

2.5. ☆ Les dictionnaires (ou tableaux associatifs)

Un dictionnaire est une collection modifiable de couples `<clé non modifiable, valeur modifiable>` permettant un accès à la valeur si on fournit la clé. On peut le voir comme une liste dans laquelle l'accès à un élément se fait par un code au lieu d'un indice. L'accès à un élément est optimisé en Python.

Pour ajouter des valeurs à un dictionnaire il faut indiquer une clé ainsi qu'une valeur :

```
>>> Mon1Dico = {} # dictionnaire vide, comme L=[] pour une liste
>>> Mon1Dico = dict() # idem, comme L=list() pour une liste
>>> Mon1Dico["a"] = 1
>>> Mon1Dico["b"] = 2
>>> Mon1Dico["toto"] = 3
>>> print(Mon1Dico)
{'a': 1, 'b': 2, 'toto': 3}

>>> Mon2Dico = {'a':1, 'b':2, 'toto':3}
>>> print(Mon2Dico)
{'a': 1, 'b': 2, 'toto': 3}

>>> Mon3Dico = dict( [ ('a',1) , ('b',2) , ('toto',3) ] )
>>> print(Mon3Dico)
{'a': 1, 'b': 2, 'toto': 3}
```

On peut utiliser **tout objet non modifiable comme clé**, typiquement une chaîne de caractère comme dans l'exemple précédent ou encore des tuples comme lors de l'utilisation de coordonnées :

```
>>> b = {} # dictionnaire vide
>>> b[(3,2)] = 12 # on ajoute un élément de clé (3,2) et valeur 12
>>> b[(4,5)] = 13 # on ajoute un élément de clé (4,5) et valeur 13
>>> print(b)
{(3, 2): 12, (4, 5): 13}
```

Même si l'on peut utiliser `MonDico["label"]`, cela génère une erreur si `label` n'est pas une clé présente dans le dictionnaire. La méthode `get` offre une alternative en permettant de récupérer une valeur d'un dictionnaire et, en cas d'absence de la clé, de spécifier une valeur par défaut à retourner :

```
>>> ficheFG = {}
>>> ficheFG = {"nom": "Faccanoni", "prenom":"Gloria", "batiment": "M", "bureau": 117}
>>> print(ficheFG["nom"])
Faccanoni
>>> print(ficheFG.get("nom"))
Faccanoni
>>> # print(ficheFG["telephone"]) # Error
>>> print(ficheFG.get("telephone", "Numéro inconnu"))
Numéro inconnu
```

Pour vérifier la présence d'une clé on utilise `in` :

```
>>> a = {}
>>> a["nom"] = "Engel"
>>> a["prenom"] = "Olivier"
>>> vf = "nom" in a
>>> print(vf)
True
>>> vf = "age" in a
>>> print(vf)
False
```

Il est possible de supprimer une entrée en indiquant sa clé, comme pour les listes :

```
>>> a = {}
>>> a["nom"] = "Engel"
>>> a["prenom"] = "Olivier"
>>> print(a)
{'nom': 'Engel', 'prenom': 'Olivier'}
>>> del a["nom"]
>>> print(a)
{'prenom': 'Olivier'}
```

- Pour récupérer la liste des clés on utilise la méthode `keys`
- Pour récupérer la liste des valeurs on utilise la méthode `values`
- Pour récupérer la liste des tuples contenant les clés et les valeurs en même temps, on utilise la méthode `items` qui retourne un tuple.

On peut alors créer des listes qui contiennent toutes les clés ou toutes les valeurs ou tous les couples clés/valeurs (on verra au chapitre 5 l'utilisation des listes en compréhension) :

```
>>> fiche = {"nom":"Engel","prenom":"Olivier"}
>>> # liste des clés
>>> print( [cle for cle in fiche.keys()] )
['nom', 'prenom']
>>> # liste des valeurs
>>> print( [valeur for valeur in fiche.values()] )
['Engel', 'Olivier']
>>> # liste de tuples
>>> print( [cv for cv in fiche.items()] )
[('nom', 'Engel'), ('prenom', 'Olivier')]
```

Comme pour les listes, pour créer une copie indépendante utiliser la méthode `copy` :

```
>>> d = {"k1":"Olivier", "k2":"Engel"}
>>> e = d.copy()
>>> print(d)
{'k1': 'Olivier', 'k2': 'Engel'}
>>> print(e)
{'k1': 'Olivier', 'k2': 'Engel'}
>>> d["k1"] = "XXX"
>>> print(d)
{'k1': 'XXX', 'k2': 'Engel'}
>>> print(e)
{'k1': 'Olivier', 'k2': 'Engel'}
```

2.6. ☆ Les ensembles

Les ensembles représentent une collection modifiable de valeurs immuables et distinctes non ordonnées.

Contrairement aux listes ou aux tuples, les ensembles ne permettent pas plusieurs occurrences du même élément et ne conservent pas un ordre défini. Cette caractéristique les rend très efficaces pour éliminer les doublons des listes ou des tuples, même si cela implique la perte de l'ordre original. De plus, ils sont particulièrement utiles pour effectuer des opérations mathématiques telles que les unions et les intersections.

Vous pouvez initialiser un ensemble vide à l'aide de `set()` ou avec des valeurs en lui passant une liste :

```
emptySet = set()
ECUE_MATHS_L1_S1 = set(['M11','M12','M13','M14','I11','P111'])
ECUE_INFO_L1_S1 = set(['M11','M12','I11','I12','P111'])
ECUE_PC_L1_S1 = set(['M11','I11','P111','C111'])
ECUE_SI_L1_S1 = set(['M11','M12','I11','P111'])
print(emptySet)
print(ECUE_MATHS_L1_S1)
print(ECUE_INFO_L1_S1)
print(ECUE_PC_L1_S1)
print(ECUE_SI_L1_S1)
```

```
set()
{'I11', 'M12', 'P111', 'M14', 'M13', 'M11'}
{'I11', 'M12', 'I12', 'P111', 'M11'}
{'C111', 'P111', 'M11', 'I11'}
{'M12', 'M11', 'P111', 'I11'}
```

Lorsque vous examinez la sortie des variables, veuillez noter que les valeurs dans les ensembles ne suivent pas l'ordre d'ajout. En effet, **les ensembles ne sont pas ordonnés.**

Nota bene : les accolades peuvent uniquement être utilisées pour initialiser un ensemble avec des valeurs, car l'utilisation d'accolades sans valeurs est l'un des moyens d'initialiser un dictionnaire, et non pas un ensemble. Cependant, les ensembles contenant des valeurs peuvent également être initialisés à l'aide d'accolades.

```
ECUE_MATHS_L1_S1 = {'M11','M12','M13','M14','I11','P111'}
print(ECUE_MATHS_L1_S1)
```

```
{'P111', 'M14', 'I11', 'M13', 'M12', 'M11'}
```

Pour ajouter ou supprimer des valeurs d'un ensemble, vous devez d'abord initialiser un ensemble. Vous pouvez utiliser la méthode `add` pour ajouter une valeur à un ensemble .

```
ECUE_MATHS_L1_S1.add('MDM')
print(ECUE_MATHS_L1_S1)
```

```
{'P111', 'MDM', 'M14', 'I11', 'M13', 'M12', 'M11'}
```

Nota bene : **un ensemble ne peut contenir que des objets immuables (comme une chaîne ou un tuple)**. Si on essaye d'ajouter une liste à un ensemble on obtient une `TypeError`.

Il existe plusieurs façons de supprimer une valeur d'un ensemble.

Option 1. utiliser la méthode `remove` :

```
ECUE_MATHS_L1_S1 = {'M11','M12','M13','M14','I11','P111','MCM'}
print(ECUE_MATHS_L1_S1)
ECUE_MATHS_L1_S1.remove('MCM')
print(ECUE_MATHS_L1_S1)
```

```
{'P111', 'MCM', 'M14', 'I11', 'M13', 'M12', 'M11'}
{'P111', 'M14', 'I11', 'M13', 'M12', 'M11'}
```

L'inconvénient de cette méthode est que si vous essayez de supprimer une valeur qui n'est pas dans votre ensemble, vous obtiendrez une `KeyError`.

Option 2. utiliser la méthode `discard` :

```
ECUE_MATHS_L1_S1 = {'M11','M12','M13','M14','I11','P111','MCM'}
print(ECUE_MATHS_L1_S1)
ECUE_MATHS_L1_S1.discard('MCM')
print(ECUE_MATHS_L1_S1)
```

```
{'P111', 'MCM', 'M14', 'I11', 'M13', 'M12', 'M11'}
{'P111', 'M14', 'I11', 'M13', 'M12', 'M11'}
```

L'avantage de cette approche sur la précédente est que si vous essayez de supprimer une valeur qui ne fait pas partie de l' ensemble, vous n'obtiendrez pas de `KeyError`. Cela fonctionne de manière similaire à la méthode `get` de dictionnaire.

Vous pouvez utiliser la méthode `clear` pour supprimer toutes les valeurs d'un ensemble.

```
ECUE_MATHS_L1_S1 = {'M11','M12','M13','M14','I11','P111','MCM'}
print(ECUE_MATHS_L1_S1)
ECUE_MATHS_L1_S1.clear()
print(ECUE_MATHS_L1_S1)
```

```
{'P111', 'MCM', 'M14', 'I11', 'M13', 'M12', 'M11'}
set()
```

Comme pour les listes, on peut parcourir les éléments d'un ensemble :

```
ECUE_MATHS_L1_S1 = {'M11','M12','M13','M14','I11','P111'}
for ecue in ECUE_MATHS_L1_S1:
    print(ecue, end=", ")
```

```
P111, M14, I11, M13, M12, M11,
```

Bien sûr, les valeurs imprimées ne sont pas dans l'ordre dans lequel elles ont été ajoutées (**les ensembles ne sont pas ordonnés**).

Si vous trouvez que vous devez obtenir les valeurs de votre ensemble sous une forme ordonnée, vous pouvez utiliser la fonction `sorted` qui génère une **liste** ordonnée (ici dans l'ordre alphabétique croissant).

```
ECUE_MATHS_L1_S1 = {'M11','M12','M13','M14','I11','P111'}
L = sorted(ECUE_MATHS_L1_S1)
print(L)
```

```
['I11', 'M11', 'M12', 'M13', 'M14', 'P111']
```

ATTENTION Les ensembles sont le moyen le plus rapide pour supprimer les doublons d'une liste :

```
L = [1,2,3,4,3,2,3,1,2]
print(L)
L = list(set(L))
print(L)
```

```
[1, 2, 3, 4, 3, 2, 3, 1, 2]
[1, 2, 3, 4]
```

ou pour établir si une liste contient de doublons :

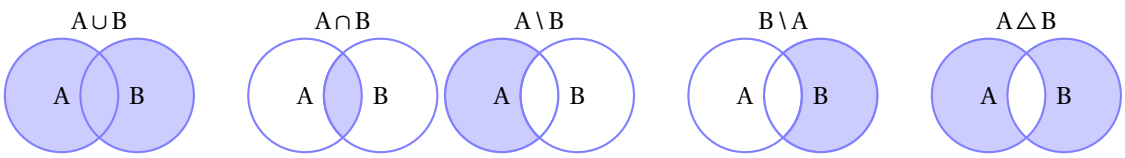
```
L = [1,2,3,4,3,2,3,1,2]
print(len(L)==len(set(L)))
```

```
False
```

Une utilisation courante des ensembles consiste à calculer des opérations mathématiques standard telles que l'union, l'intersection, la différence et la différence symétrique.

Soit E un ensemble. On note $\mathscr{P}(E)$ l'ensemble des parties de E. Soient A et B deux éléments de $\mathscr{P}(E)$. Les quatre éléments $A \cup B$, $A \cap B$, $A \setminus B$, $A \triangle B$ de $\mathscr{P}(E)$ sont définies de la façon suivante : pour tout $x \in E$,

- $x \in A \cup B \iff x \in A$ ou $x \in B$, [réunion des ensembles A et B]
- $x \in A \cap B \iff x \in A$ et $x \in B$, [intersection des ensembles A et B]
- $x \in A \setminus B \iff x \in A$ et $x \notin B$, [différence]
- $x \in A \triangle B \iff x \in A \setminus B$ ou $x \in B \setminus A$. [différence symétrique]



```
MATH = {'M11','M12','M13','M14','I11','P111'}
INFO = {'M11','M12','I11','I12','P111'}
PC = {'M11','I11','P111','C111'}
SI = {'M11','M12','I11','P111'}

union = MATH.union(INFO)                # MATH | INFO
intersection = MATH.intersection(INFO)  # MATH & INFO
MsansI = MATH.difference(INFO)
IsansM = INFO.difference(MATH)
MIsymdiff = MATH.symmetric_difference(INFO)

print(f'MATH = {MATH}')
print(f'INFO = {INFO}')
print(f'MATHS OU INFO = {union}')
print(f'MATHS ET INFO = {intersection}')
print(f'MATHS moins INFO = {MsansI}')
print(f'INFO moins MATHS = {IsansM}')
print(f'INFO Delta MATHS = {MIsymdiff}')
print(f'union moins intersection = {union.difference(intersection)}')
```

```
MATH = {'P111', 'M14', 'I11', 'M13', 'M12', 'M11'}
INFO = {'P111', 'I11', 'M12', 'M11', 'I12'}
MATHS OU INFO = {'I11', 'M12', 'I12', 'P111', 'M14', 'M13', 'M11'}
MATHS ET INFO = {'P111', 'M11', 'M12', 'I11'}
MATHS moins INFO = {'M13', 'M14'}
INFO moins MATHS = {'I12'}
INFO Delta MATHS = {'M14', 'M13', 'I12'}
union moins intersection = {'M13', 'M14', 'I12'}
```

Vous pouvez vérifier si un ensemble est un sous-ensemble d'un autre en utilisant la méthode `issubset`.

```
MATH = {'M11','M12','M13','M14','I11','P111'}
Mxx = {'M11','M12','M13','M14'}


print(f'MATH = {MATH}')
print(f'Mxx = {Mxx}')
print(f'Mxx sous-ensemble de MATH ? {Mxx.issubset(MATH)}')
```

```
MATH = {'P111', 'M14', 'I11', 'M13', 'M12', 'M11'}
Mxx = {'M13', 'M12', 'M11', 'M14'}
Mxx sous-ensemble de MATH ? True
```

2.7. Exercices

Détermination de types : `int`, `float`, `str`, `bool`, `list`, `tuple`, `range`

Pour découvrir le type d'un objet en Python, on utilise la fonction `type(obj)`. Nous avons étudié quatre types primitifs et trois structures de référence :

```
>>> a = 4; a; type(a)
4
<class 'int'>
>>> b = 4.5; b; type(b)
4.5
<class 'float'>
>>> c = "4"; c; type(c)
'4'
<class 'str'>
>>> d = True; d; type(d)
True
<class 'bool'>
```

```
>>> r = range(4); r; type(r)
range(0, 4)
<class 'range'>
>>> L = [a,b,c,d,r]; L; type(L)
[4, 4.5, '4', True, range(0, 4)]
<class 'list'>
>>> T = (a,b,c,d,r); T; type(T)
(4, 4.5, '4', True, range(0, 4))
<class 'tuple'>
```

`Typecasting` (conversions entre types)

En programmation, la conversion de type consiste à transformer des données d'un type en un autre. En Python, il existe deux types de conversion de type :

- **Conversion implicite (automatique)** : dans certaines situations, Python convertit automatiquement un type de données en un autre, sans intervention de l'utilisateur. Par exemple, si l'on additionne un `int` et un `float`, le résultat sera un `float`. Python élève automatiquement le type le plus petit vers le type le plus grand pour éviter toute perte d'information.
- **Conversion explicite (manuelle)** : dans ce cas, c'est l'utilisateur qui transforme le type d'un objet en un type spécifique. On utilise pour cela des fonctions intégrées telles que `int()`, `float()`, `str()`, etc. Cette opération est aussi appelée *typecasting*, car l'utilisateur modifie volontairement le type des données.

list ⇝ tuple

```
>>> L = [1, "cool", 1.41]
>>> tuple(L)
(1, 'cool', 1.41)
```

str ⇝ tuple

```
>>> s = "cool"
>>> tuple(s)
('c', 'o', 'o', 'l')
```

tuple ⇝ list

```
>>> T = (1, "cool", 1.41)
>>> list(T)
[1, 'cool', 1.41]
```

str ⇝ list

```
>>> s = "cool"
>>> list(s)
['c', 'o', 'o', 'l']
```

int ⇝ str

```
>>> n = 123
>>> str(n)
'123'
```

float ⇝ str

```
>>> r = -1.41
>>> str(r)
'-1.41'
```

str ⇝ int

```
>>> s = "123"
>>> int(s)
123
```

str ⇝ float

```
>>> s = "123"
>>> float(s)
123.0
>>> s = "-1.41"
>>> float(s)
-1.41
```

Il est aussi possible d'utiliser `eval()`, mais cela peut être risqué et doit être fait avec précaution.

Exercice 2.1 (Devine le résultat – ASCII art)

```
print("/\_/\ \n>^.^< \n / \ \n(___)__/")
```

Correction

```
/\_/\
>^.^<
 / \
(___)__/
```

Exercice 2.2 (Devine le résultat – affectations et print)

Soit l'instruction `A = 2+3`. Qu'afficheront les instructions suivantes ?

`print("A")` `print(A)` `print(f"A")` `print(f"{A}")` `print(f"{A = }")`

Correction

```
>>> A = 2+3
>>> print("A")
A
>>> print(A)
5
>>> print(f"A")
A
>>> print(f"{A}")
5
>>> print(f"{A = }")
A = 5
```

Exercice 2.3 (Cocorico)

Soit `coco = "rico"`. Quelle instruction affichera "`cocorico`" parmi les suivantes ?

a) `print(coco+rico)` b) `print('coco'+rico)` c) `print('coco'+coco)` d) `print(coco+'rico')`

Correction

Lorsque vous additionnez deux chaînes de caractères vous obtenez un comportement différent que lorsque vous additionnez deux nombres.

```
# print(coco+rico) # erreur car rico n'est pas une variable affectée
# print('coco'+rico) # idem
print('coco'+coco)
# print(coco+'rico') # ricorico
```

```
cocorico
```

Exercice 2.4 (Devine le résultat - String: + et *)

```
len("lo"+("la"*5+" ")*4)
```

Correction

`len` renvoie le nombre de caractères d'une chaîne :

- `"la"*5` ⇝ une chaîne de $2 \times 5 = 10$ caractères
- `"la"*5+" "` ⇝ une chaîne de 11 caractères
- `("la"*5+" ")*4` ⇝ une chaîne de $11 \times 4 = 44$ caractères
- `("la"*5+" ")*4+"lo"` ⇝ une chaîne de 46 caractères

Vérifions :

```
>>> "lo"+("la"*5+" ")*4
'lolalalalala lalalalala lalalalala lalalalala '
>>> len("lo"+("la"*5+" ")*4)
46
```

Exercice 2.5 (String concatenation)

Considérons les affectations

```
a = "six"
b = a
c = " > "
d = "ty"
e = "1"
```

Modifier les variables en utilisant exclusivement les valeurs de `a`, `b`, `c`, `d`, `e` de sort à ce que l'expression `a+c+e+b` affiche `sixty > 11 > six`. Vous pouvez utiliser l'addition de chaînes (+) et la multiplication (*) sur les chaînes.

Correction

```
a = "six"
b = a
c = " > "
d = "ty"
e = "1"
a += d # a -> 'sixty'
e *= 2 # equivalente a e = 2*e, e -> '11'
e += c # e -> '11 > '
print(a + c + e + b)
```

```
sixty > 11 > six
```

Exercice 2.6 (Devine le résultat – opérations et conversions de types)

Prédire le résultat de chacune des instructions suivantes, puis vérifier-le dans l'interpréteur Python. Attention : une instruction lève une erreur. Expliquer pourquoi.

```
print( str(4) * int("3") )
print( int("3") + float("3.2") )
print( str(3) * float("3.2") )
print( str(3/4) * 2 )
print( 3/4 * 2 )
```

Correction

```
>>> print( str(4) * int("3") ) # "4"*3
444
>>> print( int("3") + float("3.2") )  # 3*3.2
6.2
>>> # print( str(3) * float("3.2") ) # "3"*3.2, TypeError: can't multiply sequence by non-int of type
↪  'float'
>>> print( str(3/4) * 2 ) # idem que "0.75"*3.2
0.750.75
>>> print( 3/4 * 2 ) # idem que 0.75*3.2
1.5
```

Exercice 2.7 (Opérations avec différents types)

Plusieurs variables de types différents sont déjà définies.

```
monthly_savings = 100
num_months = 12
intro = "Hello"
```

Compléter le script de la manière suivante.

- Calculer le produit de `monthly_savings` (économies mensuelles) et `num_months`. Stocker le résultat dans la variable `year_savings`, qui représente les économies réalisées sur une année. Quel type de données obtenez-vous ? Vérifiez votre réponse en affichant le type de `year_savings`.

- Calculer la somme de `intro` avec lui-même et stocker le résultat dans une nouvelle variable `doubleintro`. Afficher la valeur de `doubleintro`. Le résultat correspond-il à ce que vous attendiez?

Correction

```
# Calculer le produit de monthly_savings et num_months et le stocker dans year_savings
year_savings = monthly_savings * num_months
# Imprimer le type de year_savings
print(type(year_savings))
```

```
<class 'int'>
```

```
# Calculer la somme de intro et intro et stocker le résultat dans doubleintro
doubleintro = intro + intro # idem que intro*2
# Afficher doubleintro
print(doubleintro)
```

```
HelloHello
```

Exercice 2.8 (Calculer l'age)

Affecter la variable `year` avec l'année courante et `birthyear` avec l'année de votre naissance. Afficher, à l'aide d'une f-string et de la fonction `print`, la phrase "Né en xxxx, j'ai xx ans." où xx sera calculé automatiquement. Bien noter qu'il n'y a pas d'espace avant la virgule.

Correction

Pour composer la chaîne avec une f-string:

- on écrit d'abord la phrase en français en respectant la ponctuation et en écrivant le nom des variables:

  `"Né en birthyear, j'ai age ans."`

- on ajoute ensuite un `f` avant les guillemets et on entoure les variables dont on veut afficher la valeur par des **accolades**:

  `f"Né en {birthyear}, j'ai {age} ans."`

Comparons la simplicité des f-string avec les autres méthodes:

```
year = 2021
birthyear = 2000
age = year - birthyear

s1 = f"Né en {birthyear}, j'ai {age} ans" # f-string, si python > 3.6
# s1 = "Né en {}, j'ai {} ans".format(birthyear, age)
s2 = 'Né en ' + str(birthyear) + ", j'ai " + str(age) + ' ans.' # concatenation + conversion
print(s1)
print(s2)

print('Né en', birthyear, ", j'ai", age, 'ans.') # Pb espace avant la virgule
print('Né en ', birthyear, ", j'ai ", age, ' ans.', sep="")
```

```
Né en 2000, j'ai 21 ans
Né en 2000, j'ai 21 ans.
Né en 2000 , j'ai 21 ans.
Né en 2000, j'ai 21 ans.
```

Noter qu'il faut utiliser `"` au lieu de `'` lorsqu'on a un apostrophe dans la chaîne à afficher.

Exercice 2.9 (Happy Birthday)

Soit les chaînes de caractères suivantes:

```
h = 'Happy birthday'
t = 'to you'
p = 'Prenom'
```

Imprimer la chanson *Happy birthday* par **concaténation** de ces chaînes (aller à la ligne à la fin de chaque couplet mais utiliser **une seule instruction** `print`). Le résultat devra être le suivant (bien noter les espaces et les retours à la ligne).

```
Happy birthday to you
Happy birthday to you
Happy birthday Prenom
Happy birthday to you
```

Suggestion: utiliser `f-string` (ou + et *) et \n.

Correction

```python
h = 'Happy birthday'
t = 'to you'
p = 'Prenom'
print(f"{h} {t}\n{h} {t}\n{h} {p}\n{h} {t}")
# idem que
# couplet_1 = h+' '+t+'\n'
# couplet_2 = h+' '+p+'\n'
# print(couplet_1*2+couplet_2+couplet_1)
# Suivant: ajout d'une espace avant chaque ligne
# print(h,t,"\n",h,t,"\n",h,p,"\n",h,t)
```

```
Happy birthday to you
Happy birthday to you
Happy birthday Prenom
Happy birthday to you
```

Exercice 2.10 (Compter le nombre de caractères blancs)

Écrire un script qui compte le nombre de caractères blancs `" "` contenus dans une chaîne. Par exemple, si la chaîne de caractères est `s="Bonjour le monde !"`, on devra obtenir 3. *Suggestion*: méthode `count`.

Correction

On peut utiliser la méthode `count` comme suit :

```python
s = "Bonjour le monde !"
n = s.count(" ")
print(f"La chaîne '{s}' contient {n} caractères blancs.")
# print("La chaîne '{}' contient {} caractères blancs.".format(s,n))
# print("La chaîne '" + s + "' contient " + str(n) + " caractères blancs.")
# print("La chaîne '", s, "' contient", n, "caractères blancs.") # Pb espaces
```

```
La chaîne 'Bonjour le monde !' contient 3 caractères blancs.
```

Exercice 2.11 (Nombre de chiffres de l'écriture d'un entier)

Pour $n \in \mathbb{N}$ donné, calculer et afficher le nombre de chiffres qui le composent.

Correction

1. Première idée : on transforme le nombre en chaîne de caractères (fonction `str`) et on compte la longueur de la chaîne (fonction `len`) puis on affiche le résultat (fonction `print`) :

   ```
   >>> n = 123456
   >>> print(len(str(n)))
   6
   ```

   Remarque : avec des nombres très grands, cette méthode est totalement inefficace. Pire, si le nombre a plus de 4300 chiffres, comme par exemple 7^{1000}, on ne peut pas faire une conversion en chaîne de caractères.

2. Deuxième idée : en se rappelant que $\log_{10}(10^k) = k$ pour $k \in \mathbb{N}$ et que $\log_{10}$ est une fonction croissante, on a

   $$\underbrace{\log_{10}(10^{\ell})}_{=\ell} \leq \log_{10}(n) \leq \underbrace{\log_{10}(10^{\ell+1})}_{=\ell+1}$$

   où $\ell = \mathrm{E}(\log_{10}(n))$ est la partie entière de $\log_{10}(n)$ et $(\ell+1)$ le nombre de chiffres de n (pour utiliser les fonctions $\log_{10}$ et E il faut importer le module `math` (voir à la page 267) :

   ```
   >>> import math
   >>> n = 123456
   >>> print(1+int(math.log(n,10)))
   6
   ```

3. Troisième idée : on pense “base 10” et on divise par 10 (division entière) jusqu’à ce que le nombre est supérieur ou égale à 10 (on verra au chapitre 3 la syntaxe d’une disjonction de cas et au chapitre 6 la syntaxe d’une fonction lambda, ici récursive)

```
NbChiffres = lambda n : 1 if n<10 else 1+NbChiffres(n//10)
n = 123456
print(NbChiffres(n))
```

```
6
```

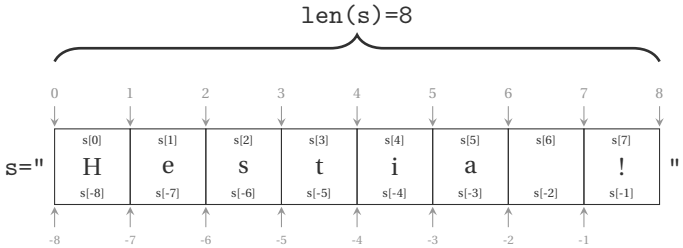
Exercice 2.12 (Devine le résultat – `string`)

Quel résultat affiche le code suivant ? Si une instruction génère une erreur, expliquer pourquoi et proposer une correction.

```
s = 'Hestia !'
print(s)
print(s*2)
print(s+" TEST")
print(s[0]) # ou s[-8]
print(s[0:])
print(s[3:])
print(s[3::1])
print(s[3::2])
print(s[3:0:-1]) #!
print(s[7:0:-1]) #!
print(s[::-1])
s[0] = 'B'
```

Héstia : déesse du foyer

Correction




```
>>> s = 'Hestia !'

>>> print(s)
Hestia !
>>> print(s*2)
Hestia !Hestia !
>>> print(s+" TEST")
Hestia ! TEST
```

```
>>> print(s[0])
H
>>> print(s[0:])
Hestia !
>>> print(s[3:])
tia !
>>> print(s[3::1])
tia !
>>> print(s[3::2])
ta!

>>> print(s[3:0:-1])
tse
>>> print(s[7:0:-1])
! aitse
>>> print(s[::-1])
! aitseH
```

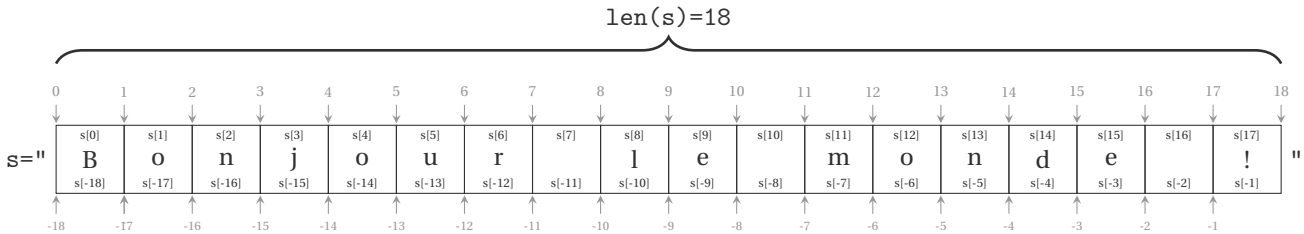
La ligne `s[0]='B'` génère une erreur car on ne peut pas modifier une chaîne. On pourra utiliser l’instruction `s='B'+s[1:]`.

Exercice 2.13 (Sous-chaînes de caractères)

On considère la chaîne de caractères `s="Bonjour le monde !"`. Déterminer les sous-chaînes suivantes : `s[:4]`, `s[6:]`, `s[1:3]`, `s[0:7:2]`, `s[2:8:3]`, `s[-3:-1]`, `s[:-4]`, `s[-5:]`, `s[-1:-3]`, `s[:-4:-1]`, `s[-5::-1]`, `s[::-1]`.

Rappel : en mode interactif il n’est pas nécessaire d’utiliser la fonction `print` (*cf.* page 13).

Correction



```
>>> s="Bonjour le monde !"
>>> s[:4]
'Bonj'
>>> s[6:]
'r le monde !'
>>> s[1:3]
'on'
>>> s[0:7:2]
'Bnor'
>>> s[2:8:3]
'nu'
>>> s[-3:-1]
'e '
>>> s[:-4]
'Bonjour le mon'
>>> s[-5:]
'nde !'
>>> s[-1:-3]
''
>>> s[:-4:-1]
'! e'
>>> s[-5::-1]
'nom el ruojnoB'
>>> s[::-1]
'! ednom el ruojnoB'
```

Exercice 2.14 (Chaîne de caractères palindrome)

Une chaîne de caractères est dite “palindrome” si elle se lit de la même façon de gauche à droite et de droite à gauche. Par exemple : `"a reveler mon nom mon nom relevera"` est palindrome (en ayant enlevé les accents, les virgules et les espaces blancs). Le plus long mot palindrome de la langue française est “ressasser”.

Pour une chaîne donnée, afficher le booléen `True` ou `False` selon que la chaîne de caractères est palindrome ou pas.

Suggestion : slicing.

Correction

```
>>> s = "bob"
>>> print(s==s[::-1])
True

>>> s = 'banana'
>>> s == s[::-1]
False
```

```
>>> s = "a reveler mon nom mon nom relevera"
>>> s = s.replace(" ","")
>>> print(s)
arevelermonnommonnomrelevera
>>> print(s==s[::-1])
True
```

Exercice 2.15 (Nombre palindrome)

Un nombre est dit “palindrome” s’il se lit de la même façon de gauche à droite et de droite à gauche. Par exemple 12321 est palindrome.

Pour $n \in \mathbb{N}$ donné, afficher le booléen `True` ou `False` selon que le nombre est palindrome ou pas. Nota bene : ne pas écrire `n="12321"`.

Suggestion : `str` et slicing.

Correction

Première méthode : on transforme le nombre en chaîne de caractères, on la retourne et on compare.

```
>>> n = 12321
>>> s = str(n)
>>> print(s==s[::-1])
True
```

```
>>> n = 123210
>>> s = str(n)
>>> print(s==s[::-1])
False
```

Remarque : pour des nombres très grands, comme 7^{1000}, la conversion en chaîne de caractères est impossible, mais on peut obtenir le même résultat en comptant combien de fois on peut diviser n par 10.

Deuxième méthode (en utilisant une boucle `while`, cf. chapitre 4) : on extrait les chiffres un par un en utilisant la division par 10 et on crée petit à petit le nouveau nombre, puis on les compare.

```
>>> n = 12321
>>> tmp = n
>>> rev_n = 0
>>> while tmp > 0:
...     tmp,digit = divmod(tmp,10)
...     rev_n = rev_n * 10 + digit
...
>>> print( n==rev_n )
True
```

```
>>> n = 123210
>>> tmp = n
>>> rev_n = 0
>>> while tmp > 0:
...     tmp,digit = divmod(tmp,10)
...     rev_n = rev_n * 10 + digit
...
>>> print( n==rev_n )
False
```

Exercice Bonus 2.16 (Tableau d'amortissement d'un prêt)

On envisage d'emprunter une somme d'argent à la banque. Avant de s'engager, il est important de comprendre comment fonctionne l'amortissement d'un prêt, notamment le coût total de l'emprunt et la répartition entre le remboursement des intérêts et du capital. Par exemple, on vérifiera qu'en empruntant 100 000 euros sur 10 ans à un taux d'intérêt annuel de 2%, on doit rembourser 110 416,14 euros, autrement dit les 100 000 euros empruntés plus 10 416,14 euros d'intérêts cumulés. Cela correspond à une mensualité de 920,13 euros et à un taux d'intérêt effectif de 10.42%.

Correction

```
Montant_Principale = 3*10**5 # euros
Taux_annuel = 5/100 # Taux d'interet annuel, dit taux nominal TAN
Duree = 20 # ans

# ====
r = Taux_annuel/12 # Taux d'interet mensuel, dit taux proportionnel
Nb_payements = Duree*12
# ====
# Payement mensuel (prêt amorti)
P = Montant_Principale * r / (1-(1+r)**(-Nb_payements))
Montant_Total = P * Nb_payements
# Le taux annuel effectif global (TAEG)
TAEG = (Montant_Total-Montant_Principale)/Montant_Principale*Duree
# Le taux équivalent (ou taux actuariel) prend en compte des intérêts composés (à savoir, le capital et
↪  les intérêts reversés pour accroître ce dernier).
Taux_equivalent = (1+Taux_annuel)**(1/12)-1

print("*"*20)
print(f"Montant de l'emprunt : {Montant_Principale} euros")
print(f"Durée du crédit : {Duree} ans")
print(f"Taux d'intérêt : {Taux_annuel*100} %")
print("*"*20)
print(f"Payement mensuel : {P:.2f} euros")
print(f"Montant total : {Montant_Total:.2f} euros")
print(f"Coût total : {Montant_Total-Montant_Principale:.2f} euros")
print(f"TAEG: {TAEG:.2f}%")
print(f"Taux équivalent: {Taux_equivalent*100:.2f}%")
print("*"*20)
```

```
********************
Montant de l'emprunt : 300000 euros
Durée du crédit : 20 ans
Taux d'intérêt : 5.0 %
********************
Payement mensuel : 1979.87 euros
Montant total : 475168.13 euros
Coût total : 175168.13 euros
TAEG: 11.68%
Taux équivalent: 0.41%
********************
```

Exercice Bonus 2.17 (Format table)

En utilisant les affectations

```
heading = '| Index of Dutch Tulip Prices |'
line = '+' + '-'*16 + '-'*13 + '+'
```

reproduire l'output

```
+-----------------------------+
| Index of Dutch Tulip Prices |
+-----------------------------+
|    Nov 23 1636 |        100 |
```

```
|    Nov 25 1636 |        673 |
|    Feb  1 1637 |       1366 |
+----------------------------+
```

Correction

```
print(line,
heading,
line,
'|    Nov 23 1636 |        100 |',
'|    Nov 25 1636 |        673 |',
'|    Feb  1 1637 |       1366 |',
line,
sep='\n')
```

```
+----------------------------+
| Index of Dutch Tulip Prices |
+----------------------------+
|    Nov 23 1636 |        100 |
|    Nov 25 1636 |        673 |
|    Feb  1 1637 |       1366 |
+----------------------------+
```

Source : https://scipython.com/book/chapter-2-the-core-python-language-i/examples/a-simple-text-table-using-python-strings/

Exercice Bonus 2.18 (Changement de casse & Co.)

Avec `s = "Un EXEMPLE de Chaîne de Caractères"`, que donneront les commandes suivantes ? Notez bien que `s` n'est pas modifiée, c'est une nouvelle chaîne qui est créée. `s.lower()`, `s.upper()`, `s.capitalize()`, `s.title()`, `s.swapcase()`, `s.center(50)`.

Correction

```
>>> s = "Un EXEMPLE de Chaîne de Caractères"
>>> s.lower() # mise en minuscules
'un exemple de chaîne de caractères'
>>> s.upper() # mise en majuscules
'UN EXEMPLE DE CHAÎNE DE CARACTÈRES'
>>> s.capitalize() # mise en majuscule de l'initiale
'Un exemple de chaîne de caractères'
>>> s.title() # mise en majuscule de l'initiale de chaque mot
'Un Exemple De Chaîne De Caractères'
>>> s.swapcase()
'uN exemple DE cHAÎNE DE cARACTÈRES'
>>> s.center(len(s)+3)
'  Un EXEMPLE de Chaîne de Caractères '
```

Exercice Bonus 2.19 (Comptage, recherche et remplacement)

Avec `s = "Monsieur Jack, vous dactylographiez bien mieux que votre ami Wolf"`, que donneront les commandes suivantes ? Une instruction lève une erreur, pourquoi ?

```
len(s) ; s.count('e') ; s.count('ie')
s.find('i') ; s.find('i',40) ; s.find('i',20,30) ; s.rfind('i')
s.index('i') ; s.index('i',40) ; s.index('i',20,30) ; s.rindex('i')
"ab;.bc;.cd".replace(';.','-'); " abcABC       ".strip();
```

Correction

```
>>> s = "Monsieur Jack, vous dactylographiez bien mieux que votre ami Wolf"
>>> len(s)
65
>>> s.count('e') # nombre de 'e'
6
>>> s.count('ie') # nombre de 'ie'
4
>>> s.find('i')  # place du premier 'i' rencontré (-1 si absent)
4
>>> s.find('i',40) # place du premier 'i' rencontré à partir de la position 40
```

```
42
>>> s.find('i',20,30) # place du premier 'i' entre 20 inclus et 30 exclu
-1
>>> s.rfind('i') # comme find, mais à rebours
59
>>> s.index('i') # comme "find" mais erreur si absent
4
>>> s.index('i',40)
42
>>> s.index('i',20,30)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> s.rindex('i') # comme index, mais à rebours
59
>>> "ab;.bc;.cd".replace(';.','-')
'ab-bc-cd'
>>> " abcABC       ".strip()
'abcABC'
```

Exercice 2.20 (Devine le résultat – Yoda)

Quel résultat donne le code suivant ?

```
txt = "Tu dois redouter le côté obscur de la Force"
# txt = "Nous sommes en danger"
# txt = "Il est ton père"

mo = txt.split(' ')
pr = mo[0].lower()
ve = mo[1]
r = ' '.join(mo[2:])
re = r[0].upper()+r[1:]
sol = ' '.join([re,pr,ve])
print(sol)
```

Correction

```
Redouter le côté obscur de la Force tu dois
```

La formation des phrases est une résultante d'une figure stylistique appelée anastrophe.

Exercice 2.21 (Classement d'une course cycliste)

On considère le tableau suivant représentant un classement de coureurs pendant une course :

```
classement = ["Sara", "Lorenzo", "Chiara", "Mirella", "Marta", "Lucia"]
```

1. Quelle expression permet d'obtenir le nombre total de coureurs ?
2. Quelle expression correspond au nom du coureur en première position ?
3. Quelle expression correspond au nom du coureur en dernière position ?
4. Quelle expression donne le nom du coureur situé à la position 3 ?

Correction

Source : https://codex.forge.apps.education.fr/exercices/course_cycliste/

```
print(f"Nombre total de coureurs : {len(classement)}")
print(f"Coureur en première position  : {classement[0]}")
print(f"Coureur en dernière position : {classement[-1]}")   # ou classement[len(classement)-1]
position = 3
print(f"Coureur en position {position} : {classement[position-1]}")
```

```
Nombre total de coureurs : 6
Coureur en première position  : Sara
Coureur en dernière position : Lucia
Coureur en position 3 : Chiara
```

La liste `classement` contient `len(classement)` = 5 éléments. Les indices vont donc de 0 à `len(classement)-1` = 4. Chaque élément de cette liste est une chaîne de caractères, représentant le prénom d'un coureur. Voici son contenu détaillé :

classement[0]	classement[1]	classement[2]	classement[3]	classement[4]	classement[5]
"Sara" str	"Lorenzo" str	"Chiara" str	"Mirella" str	"Marta" str	"Lucia" str

Exercice 2.22 (Listes et sous-listes)

On considère la liste

```
L = [ (0 ,True), "3.1415", 2, 2.0, [0.1, 0.2, "trois"] ]
```

Comment récupérer le nombre décimal (`float`) 3.1415 ? Quelle expression correspond à la chaîne (`str`) `'roi'` ? Quelle expression correspond à la chaîne (`str`) `'5'` ? Comment obtenir l'entier (`int`) 5 ?

Correction

```
>>> L = [ (0 ,True), "3.1415", 2, 2.0, [0.1, 0.2, "trois"] ]
>>> float(L[1])
3.1415
>>> L[4][2][1:4] # idem que L[-1][2][1:4]
'roi'
>>> L[1][-1] # idem que L[1][5], c'est un string
'5'
>>> int(L[1][-1]) # c'est un entier
5
```

Explications : la liste L contient `len(L)` = 5 éléments. Les indices vont de 0 à `len(L)-1` = 4. Analysons chaque élément, en indiquant en particulier son type :

L[0]	L[1]	L[2]	L[3]	L[4]
(0,True) tuple	"3.1415" str	2 int	2.0 float	[0.1, 0.2, "trois"] list

Les éléments `L[0]`, `L[1]` et `L[4]` sont des structures qu'on peut encore analyser.

Nota bene : l'élément `L[1]` n'est pas un float mais une chaîne de caractères (qui peut être transformée en un float).

Vérifions cette analyse avec les instructions suivantes (on ne s'intéresse pas ici à la syntaxe de ces instructions) :

```
>>> L = [ (0 ,True), "3.1415", 2, 2.0, [0.1, 0.2, "trois"] ]
>>> for i,x in enumerate(L):
...     print(f"L[{i}] = {x} est un objet de la {type(x)}")
...
L[0] = (0, True) est un objet de la <class 'tuple'>
L[1] = 3.1415 est un objet de la <class 'str'>
L[2] = 2 est un objet de la <class 'int'>
L[3] = 2.0 est un objet de la <class 'float'>
L[4] = [0.1, 0.2, 'trois'] est un objet de la <class 'list'>
```

- Tuple `L[0] = (0,True)`

L[0][0]	L[0][1]
0 int	True bool

  Nota bene : l'élément `L[0][1]` n'est pas une chaîne de caractères mais un booléen.

- Chaîne de caractères `L[1] = "3.1415"`

L[1][0]	L[1][1]	L[1][2]	L[1][3]	L[1][4]	L[1][5]
'3' str	'.' str	'1' str	'4' str	'1' str	'5' str

- Liste `L[4] = [0.1, 0.2, "trois"]`

L[4][0]	L[4][1]	L[4][2]
0.1	0.2	"trois"
float	float	str

Le dernier élément de cette liste peut encore être décomposé :

L[4][2][0]	L[4][2][1]	L[4][2][2]	L[4][2][3]	L[4][2][4]
't'	'r'	'o'	'i'	's'
str	str	str	str	str

Exercice 2.23 (Devine le résultat)

```
L = [ 1, 2, 3, "a", "b", "toto", "zoo", "-3.14", -3.14, [8,9,5] ]
```

```
print(L[2])
print(L[5:7])
print(L[5:])
```

```
print(L[9][2])
print(L[2]*2)
print(L[2]+2)
```

```
print(L[8]*3)
print(L[7]*3)
print(float(L[7])*3)
```

```
print(L[8]+2)
print(L[7]+2)
print(L[7]+"2")
```

Attention : certaines instructions génèrent une erreur. Expliquer la raison de ces erreurs.

Correction

```
>>> L = [1,2,3,"a","b","toto","zoo","-3.14",-3.14,[8,9,5]]
>>> print(L[2])
3
>>> print(L[5:7])
['toto', 'zoo']
>>> print(L[5:])
['toto', 'zoo', '-3.14', -3.14, [8, 9, 5]]
>>> print(L[9][2]) # L[9]=[8,9,5] et l'élément d'indice 2 de cette sous-liste est 5
5
>>> print(L[2]*2) # L[2] est un nombre
6
>>> print(L[2]+2)
5
>>> print(L[8]*3) # L[8] étant un nombre, on multiplie tout simplement
-9.42
>>> print(L[7]*3) # L[7] est un string donc on concatène 3 fois
-3.14-3.14-3.14
>>> print(float(L[7])*3) # float(L[7]) est un nombre donc on multiplie tout simplement
-9.42
>>> print(L[8]+2) # L[8] étant un nombre, on additionne tout simplement
-1.1400000000000001
>>> print(L[7]+2) # L[7] est un string, on ne peut pas lui concaténer un int
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> print(L[7]+"2") # L[7] est un string et "2" aussi, on peut les concaténer
-3.142
```

Exercice 2.24 (Saisons (liste de listes de string))

Créez quatre listes `hiver`, `printemps`, `ete` et `automne` contenant les mois correspondants à ces saisons. Créez ensuite une liste `saisons` contenant les listes `hiver`, `printemps`, `ete` et `automne`. Prévoyez ce que renvoient les instructions suivantes :

```
saisons[2]      saisons[1][0]      saisons[1:2]      saisons[:][1]
```

Correction

```
hiver = ["décembre", "janvier", "février"]
printemps = ["mars", "avril", "mai"]
```

```
ete = ["juin", "juillet", "août"]
automne = ["septembre", "octobre", "novembre"]

saisons = [hiver, printemps, ete, automne]

print(saisons[2])     # sous-liste d'index 2 (troisième sous-liste)
print(saisons[1][0]) # sous-liste d'index 2 ; puis [0] extrait le premier élément de cette sous-liste
print(saisons[1:2])  # saisons[1:2] crée une nouvelle liste contenant uniquement l'élément saisons[1],
↪  donc une liste imbriquée
print(saisons[:][1]) # saisons[:] crée une copie superficielle de la liste saisons, puis [1] sélectionne
↪  l'élément d'index 1
```

```
['juin', 'juillet', 'août']
mars
[['mars', 'avril', 'mai']]
['mars', 'avril', 'mai']
```

Exercice 2.25 (Insertions)

Considérons la liste

```
planets = ["Earth", "Mars"]
```

1. Ajouter `"Uranus"` à la fin de la liste `planets`.
2. Concaténer la liste `["Neptune", "Pluto"]` à la fin de la liste `planets`.
3. Ajouter `"Mercury"` au début de la liste `planets`.
4. Ajouter `"Venus"` entre `"Mercury"` et `"Earth"`.
5. Insérer en une seule instruction les éléments de la liste `["Jupiter", "Saturn"]` dans la liste `planets` entre `"Mars"` et `"Uranus"`.

Le résultat final doit être

```
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune', 'Pluto']
```

Suggestion: utiliser les méthodes `append`, `extend`, `insert`, `index` et le slicing.

Correction

Pour ajouter un seul élément à la fin d'une liste, on peut utiliser la méthode `append()` ou l'opérateur `+`; pour concaténer une liste à une autre liste on peut utiliser la méthode `extend()` ou l'opérateur `+`; pour insérer un (ou plusieurs) élément(s) au milieux d'une liste, on peut utiliser la méthode `insert()` ou, sinon, on extrait les deux morceaux de la liste avec un *slicing* puis on concatène les différents listes avec l'opérateur `+`:

```
planets = ["Earth", "Mars"]
print(planets)

planets.append("Uranus") # ou planets = planets + "Uranus"
print(planets)

# planets.append(["Neptune", "Pluto"]) # PB
planets.extend(["Neptune", "Pluto"]) # ou planets = planets + ["Neptune", "Pluto"]
print(planets)

planets = ["Mercury"] + planets
print(planets)

planets.insert(1,"Venus") # ou planets = planets[:1] + ["Venus"] + planets[1:]
print(planets)

idx = planets.index("Mars") # pour trouver automatiquement l'indice
# planets.insert(idx+1,["Jupiter", "Saturn"]) # PB
planets = planets[:idx+1] + ["Jupiter", "Saturn"] + planets[idx+1:]
print(planets)
```

```
['Earth', 'Mars']
['Earth', 'Mars', 'Uranus']
['Earth', 'Mars', 'Uranus', 'Neptune', 'Pluto']
```

```
['Mercury', 'Earth', 'Mars', 'Uranus', 'Neptune', 'Pluto']
['Mercury', 'Venus', 'Earth', 'Mars', 'Uranus', 'Neptune', 'Pluto']
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune', 'Pluto']
```

Il est possible d'utiliser l'opérateur augmenté +=, mais ne pas oublier la remarque de la page 43.

Exercice 2.26 (Nombres pairs)

Combien y a-t-il de nombres pairs dans l'intervalle $[2, 1000]$? *Suggestion*: utiliser les fonctions `range` et `len`.

Correction

La fonction `range` permet de générer directement les nombres pairs en spécifiant un pas de 2. Attention : pour inclure le nombre 1000, il faut écrire `range(2, 1001, 2)` car la borne supérieure de `range` est *exclusive*. Le nombre total de valeurs peut être obtenu avec `len` :

```
>>> print(len(range(2, 1001, 2)))
500
```

Ainsi, il y a 500 nombres pairs entre 2 et 1000.

Exercice 2.27 (Moyenne)

Soit `L` une liste de nombres. Calculer la somme des éléments de `L`, puis le nombre d'éléments de `L` et en déduire la moyenne arithmétique des éléments de `L`. Par exemple, si `L = [0,1,2,3]`, on doit obtenir 1.5. Attention : tester un seul exemple particulier peut être trompeur : le code peut fonctionner pour ce cas précis mais échouer sur d'autres listes. Vérifier donc votre code sur d'autres exemples dont vous connaissez le résultat.

Suggestion: utiliser les fonctions `sum` et `len`.

Correction

Pour calculer la somme des éléments de la liste, on utilise la fonction `sum()` ; pour obtenir le nombre d'éléments, on utilise la fonction `len()`.

```
>>> L = list(range(4))
>>> moy = sum(L)/len(L)
>>> print(f"L = {L}, Moyenne = {moy}")
L = [0, 1, 2, 3], Moyenne = 1.5
```

Il est important de **tester le code sur plusieurs exemples** dans des cas pour lesquels on connaît la solution. Par exemple, on peut vérifier la formule sur la liste $[1, 2, \ldots, n]$ car on sait que la moyenne est $(n+1)/2$ (en effet, on a n éléments dont la somme vaut $n(n+1)/2$). Cela permet de tester plusieurs listes juste en modifiant la valeur de n :

```
>>> n = 10
>>> L = list(range(1, n+1))
>>> moy = sum(L)/len(L)
>>> print(f"Moyenne script = {moy}, Moyenne exacte = {(n+1)/2}")
Moyenne script = 5.5, Moyenne exacte = 5.5
```

Exercice 2.28 (Effectifs et fréquence)

Soit `L` une liste de nombres entiers et soit n un élément de `L`. Calculer combien de fois n apparaît dans la liste, puis sa fréquence (effectif divisé par le nombre total d'éléments de la liste).

Exemple : si `L=[0,10,20,10,20,30,20,30,40,20]` et $n = 20$, l'effectif est 4 et la fréquence est $4/10 = 0.4$. Tester votre script sur plusieurs valeurs de n et différentes listes pour vérifier la robustesse du code.

Suggestion: utiliser `.count()` et `len()`.

Correction

Pour calculer combien de fois un élément apparaît, on utilise la méthode `count()` ; pour obtenir le nombre total d'éléments de la liste, on utilise la fonction `len()` :

```
>>> L = [0, 10, 20, 10, 20, 30, 20, 30, 40, 20]
>>> n = 20

>>> # Calcul de l'effectif
>>> effectif = L.count(n)

>>> # Calcul de la fréquence
>>> frequence = effectif / len(L)

>>> print(f"L'élément n = {n} apparaît {effectif} fois, fréquence = {frequence}")
L'élément n = 20 apparaît 4 fois, fréquence = 0.4
```

Comme pour l'exercice sur la moyenne, il est conseillé de tester le code sur plusieurs valeurs de n et sur différentes listes pour s'assurer que le calcul est correct et que le code est robuste.

Si on applique ce calcul pour chaque élément unique de L, on peut construire un histogramme de fréquences.

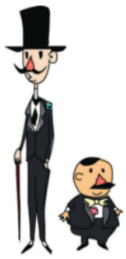
Exercice 2.29 (Max-Min)

On souhaite calculer la différence entre l'élément maximal et l'élément minimal d'une liste de nombres `L`.

- **Partie 1 : liste non vide.** Pour une liste non vide donnée, afficher la différence entre les éléments maximal et minimal. Pour cela, on pourra utiliser les fonctions `max(L)` et `min(L)`. Exemples :
  - pour `L = [1,2,3]` on affichera 2 car $3-1=2$,
  - pour `L = [5,-5]` on affichera 10 car $5-(-5)=10$,
  - pour `L = [5]` on affichera 0 car $5-5=0$.
- **Partie 2 : liste vide.** Essayez maintenant d'appliquer le même code avec une liste vide `L = []`. Que se passe-t-il ? Cela provoque une erreur, car `max` et `min` ne peuvent pas fonctionner sur une liste vide. Pour résoudre ce problème, Python permet d'utiliser le paramètre optionnel `default` :

  `min(L, default=d), max(L, default=d)`

  où `d` est la valeur retournée si la liste est vide. Corriger votre code pour qu'il marche aussi lorsque la liste est vide.



Correction

```
>>> L = [1,2,3]; print(f"L = {L}, max-min = {max(L,default=0)-min(L,default=0)}")
L = [1, 2, 3], max-min = 2
>>> L = [5,-5]; print(f"L = {L}, max-min = {max(L,default=0)-min(L,default=0)}")
L = [5, -5], max-min = 10
>>> L = [5]; print(f"L = {L}, max-min = {max(L,default=0)-min(L,default=0)}")
L = [5], max-min = 0
>>> L = []; print(f"L = {L}, max-min = {max(L,default=0)-min(L,default=0)}")
L = [], max-min = 0
>>> # L = []; print(f"L = {L}, max-min = {max(L)-min(L)}") # ValueError: max() iterable argument is
↪ empty
```

Exercice 2.30 (Range)

En utilisant l'itérateur `range`, générer et afficher les listes suivantes :

a) $A = [2,3,4,5,6,7,8,9]$ b) $B = [9,18,27,36,45,54,63,72,81]$ c) $C = [2,4,6,8,\ldots,38,40]$

d) $D = [1,3,5,7,\ldots,37,39]$ e) $E = [19,17,15,13,\ldots,3,1]$

Correction

`range(a,b,p)` génère la suite a, $a+p$, $a+2p$, …, $a+np<b$.
`range(a,b)` équivaut à `range(a,b,1)` et
`range(b)` équivaut à `range(0,b,1)`.

`range` est une itérateur (la suite n'est générée que lorsqu'on en a besoin). Pour pouvoir afficher la suite on la transforme d'abord en une liste.

```
A = range(2,10)
B = range(9,82,9)
C = range(2,41,2)
D = range(1,40,2)
E = range(19,0,-2)
```

```
print(f"A = {list(A)}")
print(f"B = {list(B)}")
print(f"C = {list(C)}")
print(f"D = {list(D)}")
print(f"E = {list(E)}")
```

```
A = [2, 3, 4, 5, 6, 7, 8, 9]
B = [9, 18, 27, 36, 45, 54, 63, 72, 81]
C = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40]
D = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39]
E = [19, 17, 15, 13, 11, 9, 7, 5, 3, 1]
```

Exercice 2.31 (Note ECUE)

Soit `CT`, `CC`, `TP` respectivement les notes de contrôle terminal, de contrôle continu et de travaux pratiques d'un ECUE. La note finale est calculée selon la formule

$$0.3\mathrm{TP} + \max\Big(\,0.7\mathrm{CT}\,,\ 0.5\mathrm{CT} + 0.2\mathrm{CC}\,\Big).$$

Écrire un script qui calcule la note finale dans les cas suivants (vérifier les résultats!) :

1. TP=10, CT=10, CC=10 ;
2. TP=10, CT=10, CC=20 ;
3. TP=10, CT=20, CC=10 ;
4. TP=20, CT=10, CC=10 ;
5. TP=20, CT=10, CC=20 ;
6. TP=20, CT=20, CC=10.

Correction

```
print( 0.3*TP+max( 0.7*CT , 0.5*CT+0.2*CC ) )
```

Cas 1 : (TP,CT,CC) = (10, 10, 10), Note = 10.0
Cas 2 : (TP,CT,CC) = (10, 10, 20), Note = 12.0
Cas 3 : (TP,CT,CC) = (10, 20, 10), Note = 17.0
Cas 4 : (TP,CT,CC) = (20, 10, 10), Note = 13.0
Cas 5 : (TP,CT,CC) = (20, 10, 20), Note = 15.0
Cas 6 : (TP,CT,CC) = (20, 20, 10), Note = 20.0

Noter que

$$0.3\mathrm{TP} + \max(0.7\mathrm{CT}, 0.5\mathrm{CT} + 0.2\mathrm{CC}) = 0.3\mathrm{TP} + 0.5\mathrm{CT} + 0.2\max(\mathrm{CT}, \mathrm{CC})$$

ainsi on aurait pu écrire

```
print( 0.3*TP + 0.5*CT + 0.2*max(CT,CC) )
```

Exercice 2.32 (texte ⇝ liste de mots)

Affiche un mot sur deux de la chaîne suivante :

```
msg = "ceci si n'est tu pas peux une lire très ceci bonne alors manière c'est de que cacher tu un
↪ t'es message trompé"
```

Correction

```
msg = """ceci si n'est tu pas peux une lire très ceci bonne alors manière c'est de que cacher tu un t'es
↪ message trompé"""
liste = msg.split(" ")
print(liste[::2])
```

```
['ceci', "n'est", 'pas', 'une', 'très', 'bonne', 'manière', 'de', 'cacher', 'un', 'message']
```

Notons que les chaînes dans la liste n'ont pas les mêmes délimiteurs!

Exercice 2.33 (Devine le résultat - tuples)

```
a = 'Quattro TT'
```

```
print( tuple(a) )
print( tuple([a]) )
print( tuple(list(a)) )
print( (a,) )
print( (a) )
print( tuple(a.split()) )
```

Correction
Les tuples vides sont construits par une paire de parenthèses vides

```
>>> empty = ()
>>> print(empty, type(empty), len(empty))
() <class 'tuple'> 0
```

Les tuples contenant un unique élément sont construits en ajoutant une virgule à l'élément (il ne suffit pas de mettre l'élément entre parenthèses) :

```
>>> s = 'hello'

>>> # t n'est pas un tuple !
>>> t = (s)
>>> print(t, type(t), len(t))
hello <class 'str'> 5
```

```
>>> # avec la virgule, t devient un tuple!
>>> t = s,
>>> print(t, type(t), len(t))
('hello',) <class 'tuple'> 1
```

Ce qu'il faut retenir : **les tuples (non vides) ne sont pas formés par les parenthèses, mais par l'utilisation de l'opérateur de virgule.**

Pour notre exercice on obtient

```
>>> a = 'Quattro TT'
>>> print( tuple(a) )   # <-- Tuple from a sequence (which is a string)
('Q', 'u', 'a', 't', 't', 'r', 'o', ' ', 'T', 'T')
>>> print( tuple([a]) ) # <-- Tuple from a sequence (which is a list containing a string)
('Quattro TT',)
>>> print( tuple(list(a)) ) # <-- Tuple from a sequence (which you create from a string)
('Q', 'u', 'a', 't', 't', 'r', 'o', ' ', 'T', 'T')
>>> print( (a,) )  # <-- Tuple containing only one element, the string
('Quattro TT',)
>>> print( (a)  )  # <-- it's just the string wrapped in parentheses
Quattro TT
>>> print( tuple(a.split()) )
('Quattro', 'TT')
```

Exercice 2.34 (Liste de tuples)
Écrire une liste contenant des tuples à deux éléments : le premier élément du tuple contient l'ingrédient à acheter, le deuxième élément contient la quantité. La liste des courses est la suivante : 2 ailes de chauve-souris, 1 paquet de beurre de limace, 10 écailles de serpent, 3 orteils de crapaud, 1 œil de triton, 8 pattes d'araignée.

Ajouter ensuite 2 langues d'escargot avec la méthode `append`. Enfin, supprimer les ailes de chauve-souris car notre magicien s'est rendu compte qu'il en avait en stock.

Correction
Comme certaines chaînes contiennent un apostrophe (ex. `pattes d'araignée`), il est nécessaire de les entourer de guillemets.

```
liste_courses = [
    ("ailes de chauve-souris", 2), ("paquet de beurre de limace", 1), ("écailles de serpent", 10),
    ("orteils de crapaud", 3),     ("œil de triton", 1),              ("pattes d'araignée", 8)
]
print(liste_courses)
```

```
[('ailes de chauve-souris', 2), ('paquet de beurre de limace', 1), ('écailles de serpent', 10),
↪  ('orteils de crapaud', 3), ('œil de triton', 1), ("pattes d'araignée", 8)]
```

Ajouter 2 langues d'escargot :

```
liste_courses.append( ("langues d'escargot", 2) )
print(liste_courses)
```

```
[('ailes de chauve-souris', 2), ('paquet de beurre de limace', 1), ('écailles de serpent', 10),
↪  ('orteils de crapaud', 3), ('œil de triton', 1), ("pattes d'araignée", 8), ("langues d'escargot",
↪  2)]
```

Supprimer les ailes de chauve-souris :

```
liste_courses.pop(0) # ou liste_courses.remove(liste_courses[0]) ou del liste_courses[0]
print(liste_courses)
```

```
[('paquet de beurre de limace', 1), ('écailles de serpent', 10), ('orteils de crapaud', 3), ('œil de
↪ triton', 1), ("pattes d'araignée", 8), ("langues d'escargot", 2)]
```

Remarques.

- `del` n'est ni une fonction ni une méthode, c'est un mot-clé (`keyword`) du langage.
- On ne peut pas simplement modifier la quantité des ailes de chauve-souris en mettant zéro avec `liste_courses[0][1]=0`, car les tuples sont immuables.
- Pour ce type de gestion d'ingrédients et de quantités, un dictionnaire (`ingredient: quantité`) serait plus pratique qu'une liste de tuples, car il permet de modifier directement les quantités sans créer de nouveaux tuples.

Exercice 2.35 (Manipulation de matrices avec des listes imbriquées)

On peut représenter une matrice avec des listes de listes. Par exemple, une matrice 3×2 peut être exprimée comme une liste contenant trois sous-listes, chacune ayant une longueur de deux.

1. Représentez la matrice
$$\mathbb{A} = \begin{pmatrix} 11 & 12 \\ 21 & 22 \\ 31 & 32 \end{pmatrix}$$
comme une liste de listes et affectez-la à la variable `A`.
2. Affichez
   - la première ligne de `A`,
   - le premier élément de la deuxième ligne,
   - le nombre de lignes de la matrice,
   - le nombre de colonnes de la première ligne.
3. Modifiez l'élément `A[2][1]` pour qu'il devienne `99` et affichez la matrice complète.

Correction

1. Représentation de la matrice
```
A = [[11, 12],
     [21, 22],
     [31, 32]]
print(A)
```
```
[[11, 12], [21, 22], [31, 32]]
```
2. En Python, les indices commencent à 0. Ainsi
```
print("Première ligne de A : ", A[0])
print("Premier élément de la deuxième ligne :", A[1][0])
print("Nombre de lignes :", len(A))
print("Nombre de colonnes :", len(A[0]))
```
```
Première ligne de A :  [11, 12]
Premier élément de la deuxième ligne : 21
Nombre de lignes : 3
Nombre de colonnes : 2
```
3. Modification d'un élément
```
A[2][1] = 99
print(A)
```
```
[[11, 12], [21, 22], [31, 99]]
```

Exercice 2.36 (Robots)

Dans cet exercice nous allons nous intéresser aux différences entre affectation simple, copie superficielle (*shallow copy*) et copie profonde (*deep copy*) pour les listes, en particulier pour des listes imbriquées.

La société **GalaxiaTech** fabrique des robots ménagers. Chaque robot est configuré à partir d'un "modèle de base" :

```
modele = ["R2List2", ["cuisson", "aspiration"], 100]
```

- `modele[0]` : nom du robot,
- `modele[1]` : liste des modules installés,
- `modele[2]` : niveau de batterie (en %).

Le développeur pressé a écrit le code suivant :

```
robot_A = modele
robot_B = modele[:]
robot_C = list(modele)

robot_A[0] = "R2List2-A"
robot_A[1].append("wifi")

robot_B[0] = "R2List2-B"
robot_B[1].append("nettoyage")

robot_C[0] = "R2List2-C"
robot_C[1].append("vidange")
robot_C[2] = 50
```

Mais catastrophe : certains robots se comportent bizarrement et le modèle de base semble avoir été "corrompu" :

```
print(f"{modele  = }")
print(f"{robot_A = }")
print(f"{robot_B = }")
print(f"{robot_C = }")
```

```
modele  = ['R2List2-A', ['cuisson', 'aspiration', 'wifi', 'nettoyage', 'vidange'], 100]
robot_A = ['R2List2-A', ['cuisson', 'aspiration', 'wifi', 'nettoyage', 'vidange'], 100]
robot_B = ['R2List2-B', ['cuisson', 'aspiration', 'wifi', 'nettoyage', 'vidange'], 100]
robot_C = ['R2List2-C', ['cuisson', 'aspiration', 'wifi', 'nettoyage', 'vidange'], 50]
```

Pourquoi tous les robots ont-ils les mêmes modules installés ? Pourquoi le modèle de base a-t-il été modifié ? Corriger le code pour que chaque robot soit totalement indépendant et que le modèle de base ne change jamais.

Correction

Le problème vient des copies directes ou des copies superficielles. Voici ce qui se passe vraiment :

- `robot_A = modele` ne crée pas de nouvelle liste mais crée un alias : toute modification affectera aussi la liste `modele` ;
- `robot_B = modele[:]` et `robot_C = list(modele)` créent une nouvelle liste au niveau supérieur, mais copient les références aux éléments (copie superficielle) : les sous-listes sont partagées, donc `robot_B[1].append("nettoyage")` modifie aussi `modele[1]` et `robot\_A[1]`, etc. ;
- seules les instructions `robot_B[0] = "R2List2-B"`, `robot_C[0] = "R2List2-C"` et `robot_C[2] = 50` n'affectent pas les autres car les entiers et les chaînes sont immuables et donc non affectés par des copies superficielles.

En effet, on peut afficher l'adresse de chaque liste et sous-liste avec l'instruction `id()` :

```
# Affichage amélioré
from tabulate import tabulate
robots = { "modele": modele, "robot_A": robot_A, "robot_B": robot_B, "robot_C": robot_C }
data = []
headers = ["objet", "id(objet)", "id([0])", "id([1])", "id([2])"]
for name, obj in robots.items():
    data.append([name, id(obj), id(obj[0]), id(obj[1]), id(obj[2])])
print(tabulate(data, headers=headers, tablefmt="grid"))
```

```
+---------+-----------------+-----------------+-----------------+-----------+
| objet   |       id(objet) |         id([0]) |         id([1]) |   id([2]) |
+=========+=================+=================+=================+===========+
| modele  | 132304525249344 | 132304524291952 | 132304525251072 |  11758856 |
+---------+-----------------+-----------------+-----------------+-----------+
| robot_A | 132304525249344 | 132304524291952 | 132304525251072 |  11758856 |
+---------+-----------------+-----------------+-----------------+-----------+
| robot_B | 132304523520256 | 132304524292016 | 132304525251072 |  11758856 |
+---------+-----------------+-----------------+-----------------+-----------+
| robot_C | 132304524296064 | 132304524292144 | 132304525251072 |  11757256 |
+---------+-----------------+-----------------+-----------------+-----------+
```

Correction avec copie profonde :

```
import copy

modele = ["R2List2", ["cuisson", "aspiration"], 100]

robot_A = copy.deepcopy(modele)
robot_B = copy.deepcopy(modele)
robot_C = copy.deepcopy(modele)

robot_A[0] = "R2List2-A"
robot_A[1].append("wifi")

robot_B[0] = "R2List2-B"
robot_B[1].append("nettoyage")

robot_C[0] = "R2List2-C"
robot_C[1].append("vidange")
robot_C[2] = 50

print(f"{modele  = }")
print(f"{robot_A = }")
print(f"{robot_B = }")
print(f"{robot_C = }")
```

```
modele  = ['R2List2', ['cuisson', 'aspiration'], 100]
robot_A = ['R2List2-A', ['cuisson', 'aspiration', 'wifi'], 100]
robot_B = ['R2List2-B', ['cuisson', 'aspiration', 'nettoyage'], 100]
robot_C = ['R2List2-C', ['cuisson', 'aspiration', 'vidange'], 50]
```

Exercice 2.37 (Pièges avec les copies et alias en Python)

Comprendre comment Python gère les variables, les listes et les copies est fondamental. Dans cet exercice, devines le résultat des différentes opérations.

1. **Variables et alias simples**

   1.1. Copie d'une variable scalaire
   ```
   a = 2
   b = a
   print(f"{a=}, {b=}")
   a = 3
   print(f"{a=}, {b=}")
   ```

   1.2. Copie et ré-affectation d'une liste
   ```
   a = [2]
   b = a
   print(f"{a=}, {b=}")
   a = [3]
   print(f"{a=}, {b=}")
   ```

   1.3. Copie et modification d'une liste
   ```
   a = [2]
   b = a
   print(f"{a=}, {b=}")
   a[0] = 3
   print(f"{a=}, {b=}")
   ```

2. **Copies de listes simples**

   2.1. Modification de la première liste
   ```
   a = list(range(5))
   b = a
   a[0] = 99
   print(a, b)
   ```

   2.2. Modification de la deuxième liste
   ```
   a = list(range(5))
   b = a
   b[0] = 99
   print(a, b)
   ```

   2.3. *Shallow copy* (copie superficielle)
   ```
   a = list(range(5))
   b = a[:]
   b[0] = 99
   print(a, b)
   ```

3. **Listes de listes et copies profondes**

3.1. Alias simple

```
a = [[1,2],[3,4]]
b = a
b[0] = 99
print(a, b)
```

3.2. *Shallow copy*

```
a = [[1,2],[3,4]]
b = a[:]
b[0] = 99
b[1][0] = 88
print(a, b)
```

3.3. *Deep copy*

```
import copy
a = [[1,2],[3,4]]
b = copy.deepcopy(a)
b[0] = 99
b[1][0] = 88
print(a, b)
```

3.4. L'opérateur `*`

```
a = [[1,2]]*5
a[0][0] = 99
print(a)
```

Remarque: utiliser `id(obj)` permet de vérifier si deux variables pointent vers le même objet en mémoire.

Correction

1. Variables et alias simples

1.1.
```
# Copie d'une variable scalaire
a = 2
b = a
print(f"Avant : {a = }, {b = }, {id(a) = }, {id(b) = }")
a = 3
print(f"Après : {a = }, {b = }, {id(a) = }, {id(b) = }")
# Remarque : les entiers sont immuables. Réaffecter a ne change pas b.
```

```
Avant : a = 2, b = 2, id(a) = 11755720, id(b) = 11755720
Après : a = 3, b = 2, id(a) = 11755752, id(b) = 11755720
```

1.2.
```
# Copie et ré-affectation d'une liste
a = [2]
b = a
print(f"Avant : {a = }, {b = }, {id(a) = }, {id(b) = }")
a = [3]
print(f"Après : {a = }, {b = }, {id(a) = }, {id(b) = }")
# Réaffectation de a => a et b pointent maintenant sur des objets différents
```

```
Avant : a = [2], b = [2], id(a) = 137914670862848, id(b) = 137914670862848
Après : a = [3], b = [2], id(a) = 137914658495424, id(b) = 137914670862848
```

1.3.
```
# Copie et modification d'une liste
a = [2]
b = a
print(f"Avant : {a = }, {b = }, {id(a) = }, {id(b) = }")
a[0] = 3
print(f"Après : {a = }, {b = }, {id(a) = }, {id(b) = }")
# Modification de l'élément d'une liste modifie aussi b, car même objet
```

```
Avant : a = [2], b = [2], id(a) = 137914657827072, id(b) = 137914657827072
Après : a = [3], b = [3], id(a) = 137914657827072, id(b) = 137914657827072
```

2. Copies de listes simples

2.1.
```
# Modification de la première liste
a = list(range(5))
b = a
print(f"Avant : {a = }, {b = }, {id(a) = }, {id(b) = }")
a[0] = 99
print(f"Après : {a = }, {b = }, {id(a) = }, {id(b) = }")
# b change aussi car alias
```

```
Avant : a = [0, 1, 2, 3, 4], b = [0, 1, 2, 3, 4], id(a) = 137914658652032, id(b) =
↪ 137914658652032
Après : a = [99, 1, 2, 3, 4], b = [99, 1, 2, 3, 4], id(a) = 137914658652032, id(b) =
↪ 137914658652032
```

2.2.
```
# Modification de la deuxième liste
a = list(range(5))
b = a
b[0] = 99
print(f"a = {a}, {b = }")
# a change aussi, même raison
```

```
a = [99, 1, 2, 3, 4], b = [99, 1, 2, 3, 4]
```

2.3.
```
# Shallow copy
a = list(range(5))
b = a[:]
print(f"Avant : {a = }, {b = }, {id(a) = }, {id(b) = }")
b[0] = 99
print(f"Après : {a = }, {b = }, {id(a) = }, {id(b) = }")
# a reste inchangé, shallow copy crée un nouvel objet
```

```
Avant : a = [0, 1, 2, 3, 4], b = [0, 1, 2, 3, 4], id(a) = 137914658651968, id(b) =
↪ 137914658652032
Après : a = [0, 1, 2, 3, 4], b = [99, 1, 2, 3, 4], id(a) = 137914658651968, id(b) =
↪ 137914658652032
```

3. Listes de listes et copies profondes

3.1.
```
# Alias simple
a = [[1,2],[3,4]]
b = a
print(f"Avant : {a = }, {b = }, {id(a) = }, {id(b) = }")
b[0] = 99
print(f"Après : {a = }, {b = }, {id(a) = }, {id(b) = }")
# Même objet => a[0] modifié
```

```
Avant : a = [[1, 2], [3, 4]], b = [[1, 2], [3, 4]], id(a) = 137914670862848, id(b) =
↪ 137914670862848
Après : a = [99, [3, 4]], b = [99, [3, 4]], id(a) = 137914670862848, id(b) = 137914670862848
```

3.2.
```
# Shallow copy
a = [[1,2],[3,4]]
b = a[:]
print(f"Avant : {a = }, {b = }, {id(a) = }, {id(b) = }")
b[0] = 99
b[1][0] = 88
print(f"Après : {a = }, {b = }, {id(a) = }, {id(b) = }")
# b[0] modifie seulement b
# b[1][0] modifie aussi a[1][0] car les sous-listes sont partagées
```

```
Avant : a = [[1, 2], [3, 4]], b = [[1, 2], [3, 4]], id(a) = 137914658651968, id(b) =
↪ 137914658495424
Après : a = [[1, 2], [88, 4]], b = [99, [88, 4]], id(a) = 137914658651968, id(b) =
↪ 137914658495424
```

3.3.
```
# Deep copy
import copy
a = [[1,2],[3,4]]
b = copy.deepcopy(a)
print(f"Avant : {a = }, {b = }, {id(a) = }, {id(b) = }")
b[0] = 99
b[1][0] = 88
print(f"Après : {a = }, {b = }, {id(a) = }, {id(b) = }")
# b est complètement indépendant de a
```

```
Avant : a = [[1, 2], [3, 4]], b = [[1, 2], [3, 4]], id(a) = 137914657106752, id(b) =
↪ 137914658651968
Après : a = [[1, 2], [3, 4]], b = [99, [88, 4]], id(a) = 137914657106752, id(b) =
↪ 137914658651968
```

3.4.
```
a = [[1,2]]*5
print(f"Avant : {a = }, {[id(x) for x in a]}")
a[0][0] = 99
print(f"Après : {a = }, {[id(x) for x in a]}")
# toutes les sous-listes pointent vers le même objet => modification répétée
```

```
Avant : a = [[1, 2], [1, 2], [1, 2], [1, 2], [1, 2]], [137914658651904, 137914658651904,
↪ 137914658651904, 137914658651904, 137914658651904]
Après : a = [[99, 2], [99, 2], [99, 2], [99, 2], [99, 2]], [137914658651904, 137914658651904,
↪ 137914658651904, 137914658651904, 137914658651904]
```

Exercice 2.38 (Devine le résultat)

```
x = list(range(1,-1,-1)) + list(range(1))
y = x[:]
z = y
z[0] = [y[k] for k in x]
x[1:3] = y[0][1:3]
z[len(y[0])-1] = 0
```

Correction

```
>>> x = list(range(1,-1,-1)) + list(range(1)) # [1,0]+[0]=[1,0,0]
>>> y = x[:] # copie superficielle
>>> z = y # z et y sont la même liste
>>> print(f"{x=},\t {id(x)=}\n{y=},\t {id(y)=}\n{z=}\t, {id(z)=}")
x=[1, 0, 0],——→ id(x)=137755346828352
y=[1, 0, 0],——→ id(y)=137755349803520
z=[1, 0, 0]——→, id(z)=137755349803520
>>> z[0] = [y[k] for k in x] # en modifiant z[0] on modifie aussi y[0]
>>> print(f"{x=},\t {id(x)=}\n{y=},\t {id(y)=}\n{z=}\t, {id(z)=}")
x=[1, 0, 0],——→ id(x)=137755346828352
y=[[0, 1, 1], 0, 0],——→ id(y)=137755349803520
z=[[0, 1, 1], 0, 0]——→, id(z)=137755349803520
>>> x[1:3] = y[0][1:3] # on ne modifie que x
>>> print(f"{x=},\t {id(x)=}\n{y=},\t {id(y)=}\n{z=}\t, {id(z)=}")
x=[1, 1, 1],——→ id(x)=137755346828352
y=[[0, 1, 1], 0, 0],——→ id(y)=137755349803520
z=[[0, 1, 1], 0, 0]——→, id(z)=137755349803520
>>> z[len(y[0])-1] = 0 # y[0]=[0,1,1], len(...)=3, z[2]=0
>>> print(f"{x=},\t {id(x)=}\n{y=},\t {id(y)=}\n{z=}\t, {id(z)=}")
x=[1, 1, 1],——→ id(x)=137755346828352
y=[[0, 1, 1], 0, 0],——→ id(y)=137755349803520
z=[[0, 1, 1], 0, 0]——→, id(z)=137755349803520
```

Exercice Bonus 2.39 (str⇝list : split et join)

Parfois il peut être utile de transformer une chaîne de caractère en liste. Cela est possible avec la méthode `split`. L'inverse est possible avec la méthode `join`.

Tester les commandes suivantes :

```
s = "Gloria:FACCANONI:Université de Toulon"
print(s)
L = s.split(":")
print(L)
print("--"*30)
L = ["Gloria","FACCANONI","Université de Toulon"]
print(L)
s = "-".join(L)
print(s)
```

Correction

```
Gloria:FACCANONI:Université de Toulon
['Gloria', 'FACCANONI', 'Université de Toulon']
------------------------------------------------------------
['Gloria', 'FACCANONI', 'Université de Toulon']
Gloria-FACCANONI-Université de Toulon
```

Exercice Bonus 2.40 (Liste de tuples et Tuple de listes)

Un tuple est non modifiable, un liste est mutable. Que se passe-t-il si on modifie un tuple de listes ? Et une liste de tuples ?

Correction

- tuple de listes

```
>>> T = (  [1,2] , ["trois","quatre"] )
>>> T[1].append("cinq") # on modifie une liste
>>> print(T)
([1, 2], ['trois', 'quatre', 'cinq'])
```

  Même si la liste a été modifiée “de l’intérieur”, Python considère que c’est toujours la même liste puisqu’elle n’a pas changé d’identifiant.

- liste de tuples

```
>>> L = [  (1,2) , ("trois","quatre") ]
>>> L[1].append("cinq") # on essaye de modifier un tuple, c'est interdit
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> print(L)
[(1, 2), ('trois', 'quatre')]
```

CHAPITRE 3

Déroulement d'un programme : instructions conditionnelles

Le *déroulement d'un programme* correspond à l'ordre dans lequel les lignes de code sont exécutées. Certaines lignes peuvent être lues une seule fois, d'autres plusieurs fois, et certaines peuvent être ignorées : tout dépend de la logique du programme. Dans ce chapitre, nous allons découvrir comment contrôler ce déroulement grâce aux **instructions conditionnelles**. Ces instructions permettent d'exécuter des blocs de code uniquement si certaines conditions sont remplies, c'est-à-dire si elles sont évaluées à `True`.

Pour comprendre l'exécution d'un code pas à pas on pourra utiliser : Visualize code and get live help http://pythontutor.com/visualize.html

3.1. Définir des conditions avec les instructions `if/elif/else`

En programmation, l'instruction `if` permet d'exécuter un bloc de code uniquement si une condition est remplie. Rappel : un bloc de code en Python est identifié par l'indentation (*cf.* section 1.3) et une condition renvoie toujours un booléen (*cf.* section 1.7).

Python propose **trois** formes principales d'instructions conditionnelles. Bien noter le rôle essentiel de l'**indentation** qui permet de délimiter chaque bloc d'instructions et la présence des **deux points** après la condition du choix (mot clé `if` et mot clé `elif`) et après le mot clé `else`.

L'instruction `if`.

```
if condition :
    # bloc exécuté si condition True
```

Exemple :

```
print("1. Avant le if")
if condition:
    print("3. Bloc if")
print("4. Après le if")
```

Output si `condition = True`

1. Avant le if
3. Bloc if
4. Après le if

Output si `condition = False`

1. Avant le if
4. Après le if

L'instruction `if...else`.

```
if condition :
    # bloc si True
else:
    # bloc si False
```

Exemple :

```
print("1. Avant le if")
if condition:
    print("3. Bloc if")
else:
    print("5. Bloc else")
print("6. Après le if")
```

Output si `condition = True`

1. Avant le if
3. Bloc if
6. Après le if

Output si `condition = False`

1. Avant le if
5. Bloc else
6. Après le if

L'instruction `if...elif...else`.

```
if condition1:
    # bloc 1
elif condition2:
    # bloc 2
else:
    # bloc 3
```

Exemple:

```
print("1. Avant le if")
if condition1:
    print("3. Bloc 1")
elif condition2:
    print("5. Bloc 2")
else:
    print("7. Bloc 3")
print("8. Après le if")
```

Output si

```
condition1 = True
condition2 = True
```

1. Avant le if
3. Bloc 1
8. Après le if

Output si

```
condition1 = False
condition2 = True
```

1. Avant le if
5. Bloc 2
8. Après le if

Importance de l'ordre des conditions. Les instructions `if/elif/else` sont évaluées *séquentiellement* : Python teste chaque condition dans l'ordre. Dès qu'une condition est vraie, le bloc correspondant est exécuté et toutes les conditions suivantes sont ignorées. Pour illustrer, considérons une fonction définie par morceaux :

$$y = \begin{cases} x & \text{si } x \leq -5, \\ 100 & \text{si } -5 < x \leq 0, \\ x^2 & \text{si } 0 < x < 10, \\ x-2 & \text{sinon.} \end{cases}$$

Code:

```
if x <= -5:
    y = x
elif x <= 0:
    y = 100
elif x < 10:
    y = x**2
else:
    y = x - 2
```

Output:

Avec x = -10 on a y = -10
Avec x = -1 on a y = 100
Avec x = 5 on a y = 25
Avec x = 15 on a y = 13

Le code commence par tester si $x \leq -5$. Si c'est vrai, il exécute le bloc correspondant et *ignore toutes les conditions suivantes.*
Si la première condition est fausse, il passe à la deuxième condition $x \leq 0$. Si cette condition est vraie, il exécute son bloc et ignore les suivantes. Ici, on n'a pas besoin d'écrire $-5 < x \leq 0$, car arriver à ce test implique déjà que $x > -5$. On continue ainsi pour toutes les conditions.

Remarque 10
La condition la plus restrictive doit être testée en premier. Par exemple :

```
if x < 10:
    y = "petit"
elif x < 0:
    y = "très petit"
else:
    y = "grand"
```

Pour $x = -5$, le premier test $x < 10$ est vrai, et le bloc `elif` ne sera jamais exécuté. Ainsi, il est important de tester d'abord les cas les plus spécifiques, puis les cas plus larges.

Imbrication. On peut utiliser une instruction `if` à l'intérieur d'une autre instruction `if`, c'est ce qu'on appelle une **instruction imbriquée** :

```
if condition1 :
    # Bloc externe
    if condition2 :
        # Bloc interne
```

Combiner plusieurs conditions. On peut combiner des conditions à l'aide des connecteurs logiques (voir à la page 19) : `and` (toutes les conditions doivent être vraies,), `or` (au moins une condition doit être vraie), `not` (inverse la valeur booléenne d'une condition.).

Remarque 11
Si a est un booléen, l'instruction `if a == True:` peut-être simplifiée en `if a:`.

Vérifier plusieurs conditions avec `all()` et `any()`. Python propose deux fonctions utiles pour tester plusieurs conditions dans une seule instruction `if` :

- `all([cond1, cond2, ...])` renvoie `True` si *toutes* les conditions sont vraies, sinon `False`.
- `any([cond1, cond2, ...])` renvoie `True` si *au moins une* condition est vraie, sinon `False`.

Ces fonctions sont très utiles pour vérifier une propriété sur des séquences ou des listes de valeurs. Par exemple, pour tester si tous les éléments d'une liste sont positifs :

```
L = [1, 4, 6, 9]
liste_de_bool = [x > 0 for x in L]
print(liste_de_bool)
print(all(liste_de_bool))
print(any(liste_de_bool))
```

```
[True, True, True, True]
True
True
```

```
L = [1, -4, 6, 9]
liste_de_bool = [x > 0 for x in L]
print(liste_de_bool)
print(all(liste_de_bool))
print(any(liste_de_bool))
```

```
[True, False, True, True]
False
True
```

⭐ Opérateur ternaire

L'opérateur ternaire est une expression qui fournit une valeur que l'on peut utiliser dans une affectation ou un calcul. Par exemple, pour trouver le minimum de deux nombres on peut utiliser l'opérateur ternaire :

```
x, y = 4, 3
# écriture classique
if x < y :
    plus_petit = x
else :
    plus_petit = y
print(f"Plus petit : {plus_petit}")
```

```
Plus petit : 3
```

```
x, y = 4, 3
# utilisation de l'opérateur ternaire
plus_petit = x if x < y else y
print(f"Plus petit : {plus_petit}")
```

```
Plus petit : 3
```

Avec l'opérateur ternaire on ne peut pas utiliser `elif`. Il faudra alors imbriquer un autre opérateur ternaire :

$$y = \begin{cases} x^2 & \text{si } x < 10 \\ x & \text{si } 10 \leq x < 20 \\ 1 & \text{sinon.} \end{cases}$$

```
# écriture classique
if x < 10 :
    y = x**2
elif x<20 :
    y = x
else :
    y = 1
```

```
# utilisation de l'opérateur ternaire
y = x**2 if x < 10 else ( x if x<20 else 1 )

# astuce mathématique pour vectorisation
y = x**2 * (x < 10) +  x * (10<=x<20) + 1 * (x>=20)
```

⭐ Traitement des erreurs — les exceptions

Afin de rendre les applications plus robustes, il est nécessaire de gérer les erreurs d'exécution des parties sensibles du code. Le mécanisme des exceptions sépare d'un côté la séquence d'instructions à exécuter lorsque tout se passe bien et, d'un autre côté, une ou plusieurs séquences d'instructions à exécuter en cas d'erreur. Lorsqu'une erreur survient, un objet exception est passé au mécanisme de propagation des exceptions, et l'exécution est transférée à la séquence de traitement ad hoc. Ce n'est pas exactement une structure de test mais elle s'en rapproche.

La syntaxe complète est la suivante : la séquence normale d'instructions est placée dans un bloc `try`. Si une erreur est détectée (levée d'exception), elle est traitée dans le bloc `except` approprié (le gestionnaire d'exception).

```
try:
    # commandes
except:
    # traitement des erreurs
finally:
    # commandes à exécuter dans tous les cas
```

Exemple :

```
a = 1
b = 0
try:
    c = a/b
    print(c)
except (ZeroDivisionError):
    print('Vous ne pouvez pas diviser par 0')
except:
    print('il y a une autre erreur')
```

```
Vous ne pouvez pas diviser par 0
```

```
a = 1
b = 'toto'
try:
    c = a/b
    print(c)
except (ZeroDivisionError):
    print('Vous ne pouvez pas diviser par 0')
except:
    print('il y a une autre erreur')
```

```
il y a une autre erreur
```

3.2. Exercices

Exercice 3.1 (Importance de l'indentation)

Les deux codes, pourtant très similaires, produisent des résultats très différents. Pourquoi ?

```
nombres = [4, 5, 6]
for nb in nombres:
    if nb == 5:
        print("Le test est vrai")
        print(f"car nb vaut {nb}")
```

```
nombres = [4, 5, 6]
for nb in nombres:
    if nb == 5:
        print("Le test est vrai")
    print(f"car nb vaut {nb}")
```

Correction

Si on observe avec attention l'indentation des instructions sur la dernière ligne, on remarque que, dans le premier, l'instruction est indentée deux fois, ce qui signifie qu'elle appartient au bloc d'instructions du test `if`. Dans le deuxième code, l'instruction n'est indentée qu'une seule fois, ce qui fait qu'elle n'appartient plus à ce bloc, d'où l'affichage de `car nb vaut xx` pour toutes les valeurs de `nb`.

```
nombres = [4, 5, 6]
for nb in nombres:
    if nb == 5:
        print("Le test est vrai")
        print(f"car nb vaut {nb}")
```

```
Le test est vrai
car nb vaut 5
```

```
nombres = [4, 5, 6]
for nb in nombres:
    if nb == 5:
        print("Le test est vrai")
    print(f"car nb vaut {nb}")
```

```
car nb vaut 4
Le test est vrai
car nb vaut 5
car nb vaut 6
```

Exercice 3.2 (Devine le résultat)

Quel résultat obtiendrez-vous en exécutant les codes suivants ? Essayez de prédire le résultat sans exécuter le code dans l'interpréteur, puis vérifiez votre prédiction en le tapant dans l'interpréteur.

Cas 1 :

```
a = 2
b = 4
if b>10:
    b = a*b
    a = b
c = a+b
print(a,b,c)
```

Cas 2 :

```
a = 2
b = 10
if b>10:
    b = a*b
    a = b
c = a+b
print(a,b,c)
```

Cas 3 :

```
a = 2
b = 13
if b>10:
    b = a*b
    a = b
c = a+b
print(a,b,c)
```

Cas 4 :

```
a = 2
b = 4
if b>10:
    b = a*b
else:
    a = b
c = a+b
print(a,b,c)
```

Cas 5 :

```
a = 2
b = 13
if b>10:
    b = a*b
else:
    a = b
c = a+b
print(a,b,c)
```

Cas 6 :

```
a = 2
b = 4
if b>10:
    b = a*b
a = b
c = a+b
print(a,b,c)
```

Correction

Cas 1 : 2 4 6 Cas 2 : 2 10 12 Cas 3 : 26 26 52 Cas 4 : 4 4 8 Cas 5 : 2 26 28 Cas 6 : 4 4 8

Exercice 3.3 (Blanche Neige)

Soient a, b et $c \in \mathbb{N}$.

Considérons le code

```
s = ""
if a<b<c:
    if 2*a<b:
        s += "Prof"
    else:
        s += "Timide"
    if 2*c<b:
        s += "Atchoum"
else:
    if a<b:
        s += "Joyeux"
    if a<c: # ♡
        s += "Simplet"
    elif b<c:
        s += "Dormeur"
    else:
        s += "Grincheux"
print(s)
```

1. Quel résultat obtient-on dans les 5 cas suivants :
   Cas 1 : $a = 1$, $b = 1$, $c = 1$
   Cas 2 : $a = 2$, $b = 1$, $c = 2$
   Cas 3 : $a = 4$, $b = 5$, $c = 2$
   Cas 4 : $a = 1$, $b = 4$, $c = 7$
   Cas 5 : $a = 4$, $b = 5$, $c = 6$
2. Trouver, s'il existe, un triplet $(a, b, c) \in \mathbb{N}^3$ tel que le code affichera `Prof` et `Timide` en même temps.
3. Trouver, s'il existe, un triplet $(a, b, c) \in \mathbb{N}^3$ tel que le code affichera `Atchoum`.
4. Même question pour `Simplet`.

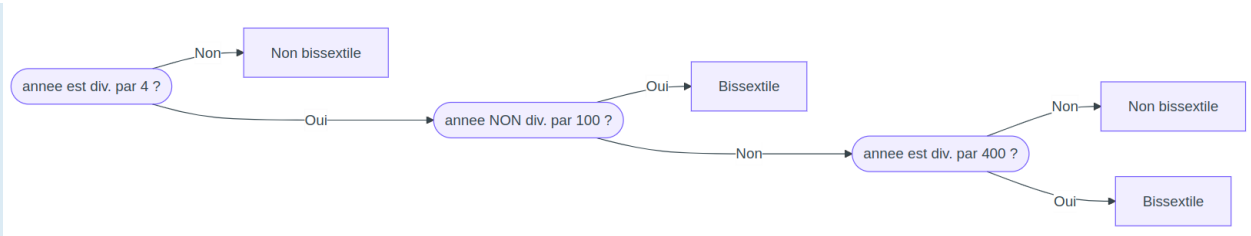
Correction

1. Notons que $a < b < c$ équivaut à "$a < b$ AND $b < c$". Sa négation (cas `else` de la ligne 8) est donc "$a \geq b$ OR $b \geq c$".
   Cas 1 : (a,b,c)= (1, 1, 1) Grincheux
   Cas 2 : (a,b,c)= (2, 1, 2) Dormeur
   Cas 3 : (a,b,c)= (4, 5, 2) JoyeuxGrincheux
   Cas 4 : (a,b,c)= (1, 4, 7) Prof
   Cas 5 : (a,b,c)= (4, 5, 6) Timide
   Notons que, si à la ligne ♡ on avait écrit `elif` au lieu de `if`, dans le cas 3 on aurait obtenu juste "Joyeux".
   Plus généralement, `Joyeux` sera toujours suivi d'un parmi `Simplet`, `Dormeur` ou `Grincheux`. Ces trois en revanche ne pourrons jamais être affichés en même temps.
2. Il n'est pas possible d'afficher `Prof` et `Timide` en même temps car pour afficher `Prof` il faut choisir a, b, c tels que $2a < b$ et pour afficher `Timide` il faut choisir a, b, c tels que $2a \geq b$.
3. Pour afficher `Atchoum` il faut que $a < b < c$ et $2c < b$, donc $b < c$ et $b > 2c$ ce qui est impossible si $b, c \geq 0$.
4. Pour afficher `Simplet` il faut que les deux conditions soient satisfaites :
$$\begin{cases} a \geq b \text{ OU } b \geq c \\ a < c \end{cases}$$
   ce qui correspond à
$$\begin{cases} a \geq b \\ a < c \end{cases} \quad \text{OU} \quad \begin{cases} b \geq c \\ a < c \end{cases}$$
   soit encore
$$b \leq a < c \quad \text{OU} \quad a < c \leq b$$
   Nous cherchons donc un triplet a, b, c tel que soit $b \leq a < c$ soit $a < c \leq b$. Notons que dans le cas $a < c \leq b$ on affichera aussi `Joyeux`. Voici deux exemples :
   Le triplet (a,b,c)= (1, 1, 3) donne Simplet
   Le triplet (a,b,c)= (1, 2, 2) donne JoyeuxSimplet

Exercice 3.4 (Années bissextiles)

Les années bissextiles comportent 366 jours. Contrairement à ce que l'on pense couramment, elles ne se rencontrent pas exactement tous les 4 ans. En effet :

- une année non divisible par 4 n'est pas bissextile,
- parmi les années divisibles par 4 :
  - les années qui ne sont pas divisibles par 100 sont bissextiles,
  - parmi les années divisibles par 100 :
    - ⋆ les années qui ne sont pas divisibles par 400 ne sont pas bissextiles,
    - ⋆ les autres sont bissextiles.



Par exemple : 2022 n'est pas divisible par 4 donc elle n'est pas bissextile ; 2020 est divisible par 4, mais pas par 100 donc elle est bissextile ; 2100 est divisible par 4, par 100, mais pas par 400 donc elle n'est pas bissextile ; 2400 est divisible par 4, par 100 et par 400 donc elle est bissextile.

Compléter le script ci-dessous affichant le booléen `True` si la variable `an` est bissextile, `False` dans le cas contraire.

```
an = 2400 # essayer les autres années des exemples
print(f"L'année {an} est-elle bissextile ?")

if an % 4 == 0 : # an div par 4
    ...
else :
    rep = False

print(rep)
```

Source : https://codex.forge.apps.education.fr/exercices/bissextile/

Correction

Si l'on applique la règle telle qu'elle est formulée dans l'énoncé on obtient

```
an = 2400
print(f"L'année {an} est-elle bissextile ?")

if an % 4 == 0 : # an div par 4
    if an % 100 != 0: # an NON div par 100
        rep = True
    else :
        if an % 400 == 0 : # an div par 400
            rep = True
        else :
            rep = False
else :
    rep = False

print(rep)
```

```
L'année 2400 est-elle bissextile ?
True
```

Soit encore

```
an = 2400
print(f"L'année {an} est-elle bissextile ?")

if an % 4 != 0 : # an NON div par 4
    rep = False
else :
    if an % 100 != 0: # an NON div par 100
        rep = True
    else :
        if an % 400 == 0 : # an div par 400
            rep = True
        else :
            rep = False

print(rep)
```

```
L'année 2400 est-elle bissextile ?
True
```

Il est aussi possible de structurer le code avec des `elif`

```
if an % 4 != 0 : # an NON div par 4
    rep = False
elif an % 100 != 0: # an div par 4 et NON div par 100
    rep = True
elif an % 400 == 0 : # an div par 4, div par 100, div par 400
    rep = True
else :
    rep = False
```

On peut aussi répondre en une ligne :

```
# if an % 4 == 0 and (an % 100 != 0 or an % 400 == 0) :
# idem que
```

```
if (an % 4 == 0 and an % 100 != 0) or an % 400 == 0 :
    rep = True
else :
    rep = False
```

Voir aussi l'exercice 5.24.

Exercice 3.5 (Détermination de l'état de l'eau en fonction de la température)

Écrire un script qui, pour une température T donnée (en degrés Celsius), affiche l'état de l'eau à cette température, c'est à dire `"solide"`, `"liquide"` ou `"gazeux"`. on prendra comme conditions les suivantes :

- si la température est strictement négative alors l'eau est à l'état solide,
- si la température est entre 0 et 100 (compris) l'eau est à l'état liquide,
- si la température est strictement supérieure à 100.

Correction

```
if T<0:
    s = "solide"
elif T<=100:
    s = "liquide"
else:
    s = "gazeux"
print(s)
```

Pour tester ce but de script, il suffit d'affecter la variable T juste avant le bloc d'instruction `if-elif-else` et ceux pour tous les cas intéressants, à savoir un cas où T est négative, un cas ou elle est entre 0 et 100 et un ou elle dépasse 100. Pour vérifier que nos conditions sont correctes, on vérifiera que pour T = 0 ou T = 100 on affiche bien la chaîne `liquide`. Versions abrégées :

```
s = "solide" if T < 0 else ( "liquide" if T <= 100 else "gazeux" )
s = "solide"*(T<0) + "liquide"*(0<=T<=100) + "gazeux"*(T>100)
```

Exercice 3.6 (Calculer $|x|$)

Afficher la valeur absolue d'un nombre x sans utiliser la fonction `abs`.

Correction

Idée 1 : On peut bien sur écrire

```
if x>=0:
    ax = x
else:
    ax = -x
print(f"|{x}| = {ax}")
```

Idée 2 : En réalité il suffit de changer x avec $-x$ si $x < 0$:

```
ax = x
if x<0:
    ax = -x
print(f"|{x}| = {ax}")
```

qu'on peut réécrire

```
ax = -x if x<0 else x
print(f"|{x}| = {ax}")
```

Idée 3 : Ou encore, sans utiliser de `if` explicite :

```
ax = (-x)*(x<0) + (x)*(x>=0)
print(f"|{x}| = {ax}")
```

Exercice 3.7 (Calcul de l'Indice de Masse Corporelle (IMC))

Écrivez un script capable de calculer l'Indice de Masse Corporelle (IMC) d'un individu en utilisant sa taille (en mètres) et sa masse (en kg). Ensuite, le script doit fournir un commentaire en fonction de l'IMC obtenu :

- Si l'IMC est inférieur à 25, affichez le commentaire : «Votre IMC est égal à XX, vous n'êtes pas en surpoids».
- Sinon, affichez : «Attention, votre IMC est égal à XX, vous êtes en surpoids».

Pour réaliser ce calcul, utilisez trois variables : `masse`, `taille`, et $\texttt{IMC}=\frac{\texttt{masse}}{\texttt{taille}\times\texttt{taille}}$.

Pour valider le script, testez-le avec différentes valeurs, en veillant à couvrir tous les scénarios possibles.

Correction

```
imc = masse/taille**2
if imc<25:
    print(f"Votre IMC est égal à {imc:.1f}, vous n'êtes pas en surpoids")
else:
    print(f"Attention, votre IMC est égal à {imc:.1f}, vous êtes en surpoids")
```

Voici deux exemples pour tester le script :
Avec masse = 60 et taille = 1.6 le script affiche «Votre IMC est égal à 23.4, vous n'êtes pas en surpoids»
Avec masse = 88 et taille = 1.6 le script affiche «Attention, votre IMC est égal à 34.4, vous êtes en surpoids»

Exercice 3.8 (Note ECUE)

On considère les notes suivantes pour un ECUE : `CT` (Contrôle Terminal), `CC` (Contrôle Continu) et `TP` (Travaux Pratiques). La note finale N est donnée par

$$N = 0.3TP + \max\{0.7CT; 0.5CT + 0.2CC\}.$$

Écrire un script qui calcule N **sans utiliser** la fonction `max()`, en testant les cas ci-dessous

	Test ①	Test ②	Test ③	Test ④	Test ⑤	Test ⑥
TP	10	10	10	20	20	20
CT	10	10	20	10	10	20
CC	10	20	10	10	20	10
Résultat N attendu	10.0	12	17.0	13.0	15	20.0

Correction

On remarque que $\max\{0.7CT; 0.5CT + 0.2CC\} = \max\{0.5CT + 0.2CT; 0.5CT + 0.2CC\} = 0.5CT + 0.2\max\{CT; CC\}$. On peut alors écrire

```
TP, CT, CC = ...
if CT >= CC:
    note = 0.3*TP + 0.7*CT
else:
    note = 0.3*TP + 0.5*CT + 0.2*CC
print(f"Note : {note}")
```

Version abrégée (une seule ligne) :

```
note = 0.3*TP + 0.5*CT + 0.2*( CT*(CT>=CC) + CC*(CT<CC) )
```

Exercice 3.9 ($ax^2 + bx + c = 0$)

Écrire un script qui, étant donnés trois réels a, b, c, détermine et stocke dans la variable `racines` la ou les solutions réelles (s'il y en a) de l'équation $ax^2 + bx + c = 0$. La variable `racines` devra contenir

- un tuple de deux valeurs si les racines sont réelles et distinctes ;
- un tuple d'un seul élément si la racine est unique ;
- un tuple vide si les racines sont complexes.

Tester le script pour les cas suivants

	Test ①	**Test ②**	**Test ③**	**Test ④**	**Test ⑤**
a	1	1	1	0	0
b	0	4	0	1	0
c	-4	4	4	2	3
RÉSULTAT ATTENDU	$(-2, 2)$	$(-2,)$	$()$	$(-2,)$	$()$

Pour calculer une racine carrée, on utilisera la propriété $\sqrt{p} = p^{1/2}$, c'est-à-dire `p**0.5` au lieu de `sqrt(p)`.

Bonus : voir l'exercice A.3 en annexe.

Correction
On distingue les cas suivants :

- $a = 0$ (équation linéaire)
  - si $b \neq 0$: une seule solution ;
  - si $b = 0$: pas de solution.
- $a \neq 0$ (équation quadratique)
  On calcule le discriminant $d = b^2 - 4ac$:
  - si $d > 0$: deux solutions réelles distinctes ;
  - si $d = 0$: une racine double ;
  - si $d < 0$: pas de racine réelle.

```
if a == 0:
    if b != 0:
        racines = ( -c/b , )    # tuple singleton
    else:
        racines = ()
else:
    d = b**2 - 4*a*c
    if d > 0:
        racines = ( (-b-d**0.5)/(2*a) ,
                    (-b+d**0.5)/(2*a) )
    elif d == 0:
        racines = ( -b/(2*a) , )
    else:
        racines = ( )
print(racines)
```

- Test 1 ⇝ racines=(-2.0, 2.0)
- Test 2 ⇝ racines=(-2.0,)
- Test 3 ⇝ racines=()
- Test 4 ⇝ racines=(-2.0,)
- Test 5 ⇝ racines=()

Remarque : les comparaisons directes sur les nombres réels (`d == 0`, `d > 0`, `d < 0`) peuvent échouer à cause des erreurs d'arrondi (par exemple lorsque les coefficients `a`, `b`, `c` sont de type `float`). Pour pallier ce problème, on utilise une petite tolérance ε :

```
eps = 1e-12
if abs(d) < eps: ...  # au lieu de d == 0
elif d > eps: ...     # au lieu de d > 0
else: ...             # au lieu de d < 0
```

Exercice 3.10 (Triangles)

Écrire un script qui, étant donnés trois nombres réels positifs a, b, c correspondant aux longueurs des trois cotés d'un triangle, affiche le type de triangle dont il s'agit parmi équilatéral, isocèle et scalène. Puis il affiche si le triangle est rectangle. Tester le script dans les cas suivants (dont la réponse est donnée) :

	Test ①	**Test ②**	**Test ③**	**Test ④**	**Test ⑤**	**Test ⑥**
a	2	3	1	1	3	1
b	4	2	1	0	4	1
c	5	2	1	-1	5	$\sqrt{2} = 2^{1/2}$
RÉSULTAT ATTENDU	scalène	isocèle	équilatéral	erreur de saisie	scalène rectangle	isocèle rectangle

Nota bene : attention à la comparaison entre `float`. Au lieu d'écrire `x==y` on utilisera `abs(x-y)<1.e-14` pour éviter des erreurs dues aux arrondis (*cf.* annexe A).

Correction
Afin d'éviter d'examiner tous les sous-cas, il est préférable de trier les côtés dès le départ.

Ensuite, on vérifie les conditions d'existence d'un triangle : le plus petit côté doit être strictement positif ; la somme des deux plus petits côtés doit être strictement supérieure au plus grand.

Puis, on classe les triangles selon deux critères :

1. **classification selon la longueur des côtés :**
   - **équilatéral** : si tous les côtés ont la même longueur,
   - **isocèle** : si exactement deux côtés sont de même longueur,
   - **scalène** : si aucun des côtés n'est de même longueur ;

   Attention à l'ordre des conditions : on commence par tester si le triangle est équilatéral, puis isocèle, puis scalène. Si l'ordre était inversé, un triangle équilatéral satisferait d'abord la condition d'être isocèle, empêchant la détection correcte du cas équilatéral.

2. **classification selon les angles:**
   - un triangle est **rectangle** si la somme des carrés des deux plus petits côtés est égale au carré du plus grand.

Remarque: ces deux classifications sont indépendantes (bien que l'on sache qu'aucun triangle équilatéral ne peut être rectangle).

```
A,B,C = sorted([a,b,c])

if A<=0 or A+B<=C :
    rep = "erreur de saisie"
else:
    # BLOC 1 : classification selon les côtés
    if abs(A-B)<1.e-14 and abs(B-C)<1.e-14:  # A==B==C
        rep = "équilatéral"
    elif abs(A-B)<1.e-14 or abs(B-C)<1.e-14: # A==B or B==C
        rep = "isocèle"
    else :
        rep = "scalène"

    # BLOC 2 : vérification de l'angle droit
    if abs(A**2+B**2-C**2)<1.e-14:  # A**2 + B**2 == C**2
        rep += ", rectangle"
print(rep)
```

Notons que la comparaison `A==B==C` doit s'écrire, pour des flottants, comme `abs(A-B)<1.e-14 and abs(B-C)<1.e-14`. Cependant, sachant que $C > B > A > 0$, cette condition est équivalente à `(C-B)<1.e-14`. Même remarque pour le cas isocèle.

(2, 4, 5) ⇝ scalène	(1, 1, 1) ⇝ équilatéral	(3, 4, 5) ⇝ scalène, rectangle
(3, 2, 2) ⇝ isocèle	(1, 0, -1) ⇝ erreur de saisie	(1, 1, 1.41421) ⇝ isocèle, rectangle

Exercice Bonus 3.11 (Pierre Feuille Ciseaux)

Écrire un script où le joueur entre un mot parmi "pierre", "feuille", "ciseaux", puis l'ordinateur choisit au hasard un de ces mots et il affiche le résultat ("Perdu", "Gagné", "Égalité").

Pour que l'ordinateur choisisse aléatoirement on écrira

```
import random
valide = ["pierre", "feuille", "ciseaux"]
cpu = random.choice(valide)
```

Pour affecter à la variable "`user`" le mot que le joueur tape au clavier on écrira

```
user = input("écrit ton choix: ")
```

Correction

```
import random
valide = ["pierre", "feuille", "ciseaux"]

user = input("Écrit ton choix: ")

if user not in valide:
    print("input incorrect")
else:
    cpu = random.choice(valide)
    print(f"Choix cpu: {cpu}")
    if cpu == user:
        output = "Égalité"
    elif cpu == "pierre":
        output =  "Tu as gagné" if user == "feuille" else "Tu as perdu"
    elif cpu == "ciseaux":
        output = "Tu as gagné" if user == "pierre" else "Tu as perdu"
```

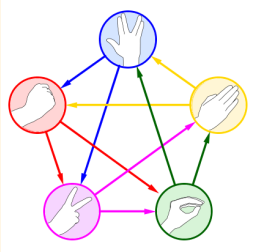
```
    else:
        output = "Tu as gagné" if user =="ciseaux" else "Tu as perdu"
    print(output)
```

Exercice Bonus 3.12 (Pierre, feuille, ciseaux, lézard, Spock)

Une variante de "Pierre feuille ciseaux" a été créée et révélée par la série américaine The Big Bang Theory et a obtenu pas mal de popularité. Créé à l'origine par Sam KASS et Karen BRYLA, elle est largement utilisé par Sheldon, Leonard, Howard et Raj. [a]

Voici les règles et toutes les combinaisons permettant de remporter (ou perdre) la partie :

- la pierre casse les ciseaux et écrase le lézard ;
- les ciseaux décapitent le lézard et coupent la feuille ;
- le lézard mange la feuille et empoisonne Spock ;
- la feuille désapprouve Spock et recouvre la pierre ;
- Spock vaporise la pierre et écrabouille les ciseaux.



Écrire un script qui joue à cette variante comme pour l'exercice 3.11.

a. Un article fort intéressant sur la construction d'autres variantes http://eljjdx.canalblog.com/archives/2015/10/21/32803533.html
Un autre article : https://info.blaisepascal.fr/nsi-chifoumi

Correction

```
import random
valide = ["pierre", "feuille", "ciseaux", "lezard", "Spock"]

user = input("Écrit ton choix: ")

if user not in valide:
    print("input incorrect")
else:
    cpu = random.choice(valide)
    print(f"Choix cpu: {cpu}")
    if user == cpu:
        output = "Égalité"
    elif user == "pierre":
        output = "Tu as gagné" if cpu == "ciseaux" or cpu=="Spock" else "Tu as perdu"
    elif user == "ciseaux":
        output = "Tu as gagné" if cpu == "lezard" or cpu=="feuille" else "Tu as perdu"
    elif user == "lezard":
        output = "Tu as gagné" if cpu == "feuille" or cpu=="Spock" else "Tu as perdu"
    elif user == "feuille":
        output = "Tu as gagné" if cpu == "Spock" or cpu=="pierre" else "Tu as perdu"
    else:
        output = "Tu as gagné" if cpu == "pierre" or cpu=="ciseaux" else "Tu as perdu"
    print(output)
```

CHAPITRE 4

Structures itératives

S'il est vrai que compter les moutons aide à s'endormir, c'est parce que la répétition pure finit par lasser l'esprit. Pour un développeur, réécrire inlassablement les mêmes lignes de code produit le même effet : l'ennui s'installe et les erreurs s'invitent. Heureusement, les langages de programmation automatisent ces tâches grâce aux **boucles**. On distingue deux familles de répétitions. On distingue traditionnellement deux approches selon la visibilité que l'on a sur la fin du processus.

- **La répétition inconditionnelle :** on sait à l'avance combien de fois le bloc d'instructions doit s'exécuter (par exemple pour chaque élément d'une liste).
- **La répétition conditionnelle :** on ignore le nombre d'itérations nécessaires ; le bloc se répète tant qu'une condition logique reste vraie.

Pour comprendre l'exécution d'un code pas à pas on pourra utiliser : http://pythontutor.com/visualize.html

4.1. Boucle `for` : répétition déterminée (parcourir)

Pour répéter un bloc d'instructions un nombre précis de fois, on utilise un *compteur actif* : une variable qui suit le nombre de répétitions. La variable `target` prend successivement les valeurs de `sequence` (liste, tuple, chaîne, dictionnaire...). À chaque itération, le bloc indenté s'exécute pour la valeur courante de `target`.

La boucle `for`.

```
for target in sequence:
    instruction_1
    instruction_2
    ...
```

✧

Exemple simple :

```
for i in range(3):
    print("i =", i)
print("Boucle terminée")
```

Output :

```
i = 0
i = 1
i = 2
Boucle terminée
```

✧

Dans l'exemple suivant, on montre l'usage d'un compteur pour générer des valeurs et remplir une liste.

Formule :

$$\left[1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}\right]$$

Code :

```
L = [] # liste vide
for n in range(4):
    L.append(1/2**n)
print(L)
```

Output :

```
[1.0, 0.5, 0.25, 0.125]
```

✧

Parcours d'un itérable On peut parcourir les éléments d'une liste, d'un tuple ou d'une chaîne soit par indice, soit directement par élément :

- **Liste**

... par indice :

```
L = ["Wow", 2, (1,0)]
for i in range(len(L)):
    print(L[i])
```

... par élément :

```
L = ["Wow", 2, (1,0)]
for item in L:
    print(item)
```

Même résultat :

```
Wow
2
(1, 0)
```

- **Chaîne**

... par indice :

```
s = "Qua"
for i in range(len(s)):
    print(s[i])
```

... par caractère :

```
s = "Qua"
for c in s:
    print(c)
```

Même résultat :

```
Q
u
a
```

Astuce : l'itération directe sur les éléments est souvent plus rapide.

Boucles imbriquées On peut placer une boucle dans une autre :

```
for x in [10,20,30,40]:
    for y in [3,7]:
        print(x+y,end=", ")
```

Pour chaque valeur de x, la boucle intérieure parcourt toutes les valeurs de y, ainsi on obtient :

```
13, 17, 23, 27, 33, 37, 43, 47,
```

Parcours de deux (ou plus) itérables On peut parcourir directement les éléments des deux (ou plusieurs) itérables simultanément soit par indice, soit directement par élément en utilisant la fonction `zip`.

- **Liste**

... par indice :

```
A = [10, 20, 30]
B = [3, 7, 9]
m = min(len(A),len(B))
for i in range(m):
    print(A[i], B[i])
```

... par élément :

```
A = [10, 20, 30]
B = [3, 7, 9]

for a, b in zip(A, B):
    print(a, b)
```

Même résultat :

```
10 3
20 7
30 9
```

- **Chaîne**

... par indice :

```
A = "Topo"
B = "Gatto"
m = min(len(A),len(B))
for i in range(m):
    print(A[i], B[i])
```

... par élément :

```
A = "Topo"
B = "Gatto"

for a, b in zip(A, B):
    print(a, b)
```

Même résultat :

```
T G
o a
p t
o t
```

L'utilisation de `zip` est plus concise et lisible, surtout pour plusieurs itérables. Elle évite de gérer explicitement les indices.

`enumerate` Pour accéder à la fois à l'indice et à l'élément :

```
L = ["Riri", "Fifi", "Loulou"]
for idx,item in enumerate(L):
    print(item, idx)
```

```
Riri 0
Fifi 1
Loulou 2
```

On peut changer l'indice de départ :

```
L = ["Riri", "Fifi", "Loulou"]
for idx,item in enumerate(L,start=101):
    print(idx, item)
```

```
101 Riri
102 Fifi
103 Loulou
```

Et le parcours d'un dictionnaire ? 4 types de parcours :

... par les clés (défaut) :

```
d = {"a": 1, "b": 2}
for k in d.keys():
# for k in d:
    print(k)
```

```
a
b
```

... par les valeurs :

```
for v in d.values():
    print(v)
```

```
1
2
```

... par paires (clé, valeur) :

```
for k, v in d.items():
    print(k, "->", v)
```

```
a -> 1
b -> 2
```

4.2. Boucle `while` : répétition indéterminée (conditionnelle)

Contrairement à la boucle `for`, qui répète un bloc un nombre déterminé de fois, la boucle `while` continue tant qu'une condition est vraie.

La boucle `while`.

```
while condition:
    instruction_1
    instruction_2
    ...
```

`while` signifie « tant que ». Tant que `condition` est `True`, le bloc d'instructions est exécuté. Dès que la condition devient `False`, la boucle s'arrête.

Attention Si la condition ne devient jamais fausse, la boucle devient infinie et le programme ne se termine pas.

Exemple simple :

```
i = 0
while i < 3:
    print("i =", i)
    i += 1
print("Boucle terminée")
```

Output :

```
i = 0
i = 1
i = 2
Boucle terminée
```

Les deux codes suivants produisent le même résultat : le premier avec une boucle `for`, le second avec une boucle `while`.

Boucle `for` :

```
for i in range(2, 10, 3):
    print(i, end=", ")
```

```
2, 5, 8,
```

Boucle `while` équivalente :

```
i = 2
while i < 10:
    print(i, end=", ")
    i += 3
```

```
2, 5, 8,
```

Attention Toute boucle `for` peut être réécrite à l'aide d'une boucle `while`, mais l'inverse n'est pas vrai : certaines boucles `while` n'ont pas de borne connue à l'avance et ne peuvent donc pas être remplacées par un `for`.

Un exemple de recherche de seuil : trouver le plus petit n tel que $1+2+\cdots+n > 10^6$.

Code :

```
n = 0
somme = 0
while somme < 10**6:
    n += 1
    somme += n
print(f"{n = }, {somme = }")
print(f"{n-1 = }, {somme-n = }")
```

Output :

```
n = 1414, somme = 1000405
n-1 = 1413, somme-n = 998991
```

4.3. Suites

En analyse numérique, une place très importante est occupée par la construction de suites $(u_n)_n$, souvent définies par récurrence. Ces suites apparaissent naturellement lors de la discrétisation de problèmes continus, par exemple dans la recherche de limites ou l'approximation numérique de solutions d'équations différentielles. On rencontrera ces types de suites notamment dans le cours DSSC-6, consacré au calcul de solutions approchées d'équations différentielles ordinaires (et plus précisément de problèmes de Cauchy).

Deux types de problèmes se rencontrent généralement :

- **Recherche de seuil.** On cherche le plus petit indice n tel que u_n satisfasse une condition donnée. Le nombre d'itérations n'étant pas connu à l'avance, l'utilisation d'une boucle `while` est nécessaire : la boucle doit continuer **tant que le critère n'est pas satisfait**.
- **Construction avec taille connue.** Lorsque le nombre d'éléments de la suite est connu à l'avance (par exemple pour des indices $n = 0 \ldots \mathrm{N}$), on peut *préallouer* la liste à la bonne taille, puis affecter ses valeurs par indice, au lieu d'utiliser la méthode `append`. Cette écriture est souvent plus claire et évite toute ambiguïté sur la taille finale de la liste.

4.3.1. Recherche de seuil

Type ① : $u_n = f(n)$ et recherche de seuil

Calculer le plus petit n tel que $u_n > s$:

```
n = ...                  # initialisation
while not( f(n) > s ):   # tant que la condition n'est pas satisfaite, i.e. f(n) <= s
    n += 1               # incrément
print(f"Le plus petit n tel que f(n)>s est {n = }")
# Vérification
print(f(n-1), s, f(n))
```

4.3.2. Suites définies par récurrence

Une grande partie des suites rencontrées en pratique peut être ramenée à l'une des quatre catégories présentées ci-dessous. Chacune correspond à un schéma algorithmique typique, qu'il est essentiel de savoir reconnaître et implémenter correctement.

Type ② : u_0 donné et $u_{n+1} = f(u_n)$

Canevas avec `append` :

```
u = [u0]   # initialisation
N = ...    # dernier indice (n = 0...N)
for n in range(0, N):
    u.append( f(u[n]) )
```

Canevas avec préallocation :

```
N = ...   # dernier indice (n = 0...N)
u = [0]*(N+1)
u[0] = u0
for n in range(0, N):
    u[n+1] = f(u[n])
```

C'est une suite à un pas : chaque terme dépend uniquement du précédent. Pour un N donné, il est essentiel de vérifier que l'algorithme calcule bien u_{N} (et non $u_{\mathrm{N}+1}$ ou $u_{\mathrm{N}-1}$), ce qui dépend directement de l'indice de la boucle.
Exemple en analyse numérique : méthode d'Euler explicite.
Curiosité : voir annexe A, exercice A.6.

◇

Type ③ : u_0, u_1 donnés et $u_{n+1} = f(u_n, u_{n-1})$

Canevas avec `append` :

```
u = [u0, u1]   # initialisation
N = ...        # dernier indice (n = 0...N)
for n in range(1, N):
    u.append( f(u[n], u[n-1]) )
```

Canevas avec préallocation :

```
N = ...   # dernier indice (n = 0...N)
u = [0]*(N+1)
u[0], u[1] = u0, u1
for n in range(1, N):
    u[n+1] = f(u[n], u[n-1])
```

Il s'agit ici d'une suite à deux pas, où chaque terme dépend des deux termes précédents. Ce type de suite apparaît naturellement dans les méthodes numériques multi-pas.
Exemple en analyse numérique : méthodes d'Adams–Bashforth.

◇

Type ④ : suite vectorielle autonome

Canevas avec `append` :

```
u, v = [u0], [v0]
N = ...  # dernier indice (n = 0...N)
for n in range(0, N):
    u.append( f_1(u[n], v[n]) )
    v.append( f_2(u[n], v[n]) )
```

Canevas avec préallocation :

```
N = ...  # dernier indice (n = 0...N)
u = [0]*(N+1)
v = [0]*(N+1)
u[0], v[0] = u0, v0
for n in range(0, N):
    u[n+1] = f_1(u[n], v[n])
    v[n+1] = f_2(u[n], v[n])
```

Exemple en analyse numérique : méthode d'Euler explicite appliquée à un système d'équations différentielles.

◇

Type ⑤ : suite vectorielle non autonome

Canevas avec `append` :

```
u, v = [u0], [v0]
N = ...  # dernier indice (n = 0...N)
for n in range(0, N):
    u.append( f_1(n, u[n], v[n]) )
    v.append( f_2(n, u[n], v[n]) )
```

Canevas avec préallocation :

```
N = ...  # dernier indice (n = 0...N)
u = [0]*(N+1)
v = [0]*(N+1)
u[0], v[0] = u0, v0
for n in range(0, N):
    u[n+1] = f_1(n, u[n], v[n])
    v[n+1] = f_2(n, u[n], v[n])
```

EXEMPLE

① On considère la suite définie par $u_n = \sqrt{n}$. Déterminer le plus petit entier n tel que $u_n > 10$.

```
n = 0
while n**0.5 <= 10 :
    n += 1
print(f"{n = }, en effet {n**0.5 = } et {(n-1)**0.5 = }")
```

```
n = 101, en effet n**0.5 = 10.04987562112089 et (n-1)**0.5 = 10.0
```

② On considère la suite définie par $u_0 = 1$ et $u_{n+1} = \frac{1}{2}u_n + 1$. Calculer les termes de la suite jusqu'à l'indice N = 10.

```
N = 10
u = [0]*(N+1)
u[0] = 1
for n in range(0, N):
    u[n+1] = 0.5*u[n] + 1
print(u)
```

```
[1, 1.5, 1.75, 1.875, 1.9375, 1.96875, 1.984375, 1.9921875, 1.99609375, 1.998046875, 1.9990234375]
```

③ On considère la suite définie par $u_0 = 1$, $u_1 = 2$ et $u_{n+1} = u_n + u_{n-1}$. Calculer les termes de la suite jusqu'à l'indice N = 7.

```
N = 7
u = [0]*(N+1)
u[0], u[1] = 1, 2
for n in range(1, N):
    u[n+1] = u[n] + u[n-1]
print(u)
```

```
[1, 2, 3, 5, 8, 13, 21, 34]
```

④ On considère les suites définies par

$$\begin{cases} u_0 = 1, \quad v_0 = 0, \\ u_{n+1} = u_n + v_n, \\ v_{n+1} = v_n - u_n. \end{cases}$$

Calculer les termes des deux suites jusqu'à l'indice N = 5.

```
N = 5
u = [0]*(N+1)
v = [0]*(N+1)
u[0], v[0] = 1, 0
for n in range(0, N):
    u[n+1] = u[n] + v[n]
    v[n+1] = v[n] - u[n]
print(u)
print(v)
```

```
[1, 1, 0, -2, -4, -4]
[0, -1, -2, -2, 0, 4]
```

⑤ On considère les suites définies par

$$\begin{cases} u_0 = 1, \quad v_0 = 1, \\ u_{n+1} = u_n + \dfrac{1}{n+1} v_n, \\ v_{n+1} = v_n - \dfrac{1}{n+1} u_n. \end{cases}$$

Calculer les termes des deux suites jusqu'à l'indice N = 5.

```
N = 5
u = [0]*(N+1)
v = [0]*(N+1)
u[0], v[0] = 1, 1
for n in range(0, N):
    u[n+1] = u[n] + v[n]/(n+1)
    v[n+1] = v[n] - u[n]/(n+1)
print(u)
print(v)
```

```
[1, 2.0, 2.0, 1.6666666666666667, 1.25, 0.8333333333333334]
[1, 0.0, -1.0, -1.6666666666666665, -2.083333333333333, -2.333333333333333]
```

4.4. ☆ Ruptures de séquences

Interrompre une boucle : `break`

L'instruction `break` interrompt immédiatement l'exécution de la boucle `for` ou `while` en cours. Elle provoque la sortie du bloc de la plus petite boucle englobante contenant cette instruction.

Exemple 1

```
for i in [1, 2, 3, 4, 5]:
    if i == 4:
        break
    print(i, end=" ")
print("Boucle interrompue pour i = ", i)
```

```
1 2 3 Boucle interrompue pour i =  4
```

Exemple 2

```
for lettre in "AbcDefGhj":
    if lettre=='D':
        break
    print(lettre, end=" ")
```

```
A b c
```

Une utilisation fréquente de `break` concerne ce que l'on appelle des boucles infinies. Il s'agit d'un type de boucle qui ne s'arrête que lorsque quelque chose se produit qui la fait cesser :

```
while True:
    ...
    if une_certaine_condition == True :
        break
```

◇

Court-circuiter une boucle : `continue`

L'instruction `continue` permet d'interrompre immédiatement l'itération en cours d'une boucle `for` ou `while`, et de passer directement à l'itération suivante. Le contrôle revient à la ligne d'en-tête de la boucle sans exécuter les instructions restantes du bloc courant.

```
for i in [1, 2, 3, 4, 5]:
    if i == 4:
        continue
    print(i, end=" ")
print("Boucle terminée")
```

```
1 2 3 5 Boucle terminée
```

L'instruction `continue` continue sur la prochaine itération de la boucle. Par exemple, supposons que nous voulions afficher $1/n$ pour n dans une liste. Si n vaut 0, le calcul $1/n$ sera impossible, il faudra donc sauter cette étape et passer au nombre suivant :

```
liste = [ -4, 2, 6, 0, 1, 3, 0, 10]
for n in liste:
    if n ==0:
        continue
    print(1/n, end=" ")
```

```
-0.25 0.5 0.16666666666666666 1.0
↪  0.3333333333333333 0.1
```

Autre exemple :

```
a=0
while a<=5:
    a+=1
    if a%2==0:
        continue
    print(a, end=" ")
print("Boucle terminée")
```

```
1 3 5 Boucle terminée
```

◇

Utilisation avancée des boucles

La syntaxe complète des boucles autorise des utilisations plus rares. Les boucles `while` et `for` peuvent posséder une clause `else` qui ne s'exécute que si la boucle se termine normalement, c'est-à-dire sans interruption.

Les instructions de boucle ont une clause `else` qui est exécutée lorsque la boucle se termine par épuisement de la liste (avec `for`) ou quand la condition devient fausse (avec `while`), mais pas quand la boucle est interrompue par une instruction `break` :

```
for item in items:
    if something(item):
        do_st(item)
        break
else: # else associé à for et non au if !!!
    print("Not found") # exécuté si la boucle se termine mais non exécuté si termine par break
```

Voici le cas typique d'un code (à gauche) et son équivalent (à droite) avec cette clause :

```
found = False
for x in l:
    if check_condition(x):
        do_something(x)
        found = True
        break
if not found:
    do_something_else()
```

```
for x in l:
    if check_condition(x):
        do_something(x)
        break
else:
    do_something_else()
```

Voici un exemple de ce besoin :

```
found = False
y = 7
x = y // 2
while x > 1:
    if y % x == 0:
        print(y, "a pour facteur", x)
        found = True
        break
    x -= 1
if not found:
    print(y, "est premier.")
```

```
7 est premier.
```

```
y = 7
x = y // 2
while x > 1:
    if y % x == 0:
        print(y, "a pour facteur", x)
        break
    x -= 1
else :
    print(y, "est premier.")
```

```
7 est premier.
```

Un exemple avec le parcours d'une liste :

```
L = [2, 5, 9, 7, 11]
cible = 5
for i in L :
    if i == cible :
        sauve = i
        break # voici l'interruption !
else : # else pour le for et non pas le if !
    print(cible, "n'est pas dans", L)
    sauve = None
# sauve vaut cible ou None
print("On obtient sauve =", sauve)
```

```
On obtient sauve = 5
```

Un exemple depuis la documentation officielle : dans la boucle suivante on recherche des nombres premiers

```
for n in range(2,10):
    for x in range(2,int(n/2)+1):
        if n%x==0:
            print(f"{n} est égale à {x}*{int(n/x)}")
            break
    else:
        print(f'{n} est un nombre premier')
```

```
2 est un nombre premier
3 est un nombre premier
4 est égale à 2*2
5 est un nombre premier
6 est égale à 2*3
7 est un nombre premier
8 est égale à 2*4
9 est égale à 3*3
```

Notons que dans de nombreux cas, on peut utiliser `any` :

```
if any(check_condition(x) for x in l):
    do_something()
else:
    do_something_else()
```

4.5. Exercices

Exercice 4.1 (Devine le résultat - `for`)

Quel résultat donnent les codes suivants ? Après avoir écrit votre réponse, vérifiez-là avec l'ordinateur.

Cas 1 :
```
L = ["a", "b"]
for elem in L:
    print(elem)
```

Cas 2 :
```
L = ["a", "b"]
for i in range(len(L)):
    print(L[i])
```

Cas 3 :
```
L = ["a", "b"]
for i in range(len(L)):
    print(i, L[i])
```

Cas 4 :
```
L = ["a", "b"]
for i,elem in enumerate(L):
    print(i, L[i])
```

Cas 5 :
```
for n in [-5, "zoo", 2.71]:
    print(n)
```

Cas 6 :
```
for i,n in enumerate(["a", -1]):
    print(f"(i,n)=({i},{n})")
```

Cas 7 :
```
for n in range(5):
    print(n)
```

Cas 8 :
```
for n in range(2,8):
    print(n)
```

Cas 9 :
```
for n in range(2,8,2):
    print(n)
```

Cas 10 :
```
for n in range(10):
    if n%4==0:
        print(n)
```

Cas 11 :
```
for n in range(10):
    if n%4==0:
        print(n)
    elif n%2==0:
        print(2*n)
    else:
        None
```

Cas 12 :
```
for n in range(10):
    if n%2==0:
        print(2*n)
    elif n%4==0:
        print(n)
    else:
        None
```

Correction

Voici les sorties (sans les retours à la ligne) :

Cas 1 : a b	Cas 5 : -5 zoo 2.71	Cas 9 : 2 4 6
Cas 2 : a b	Cas 6 : (i,n)=(0,a) (i,n)=(1,-1)	Cas 10 : 0 4 8
Cas 3 : 0 a 1 b	Cas 7 : 0 1 2 3 4	Cas 11 : 0 4 4 12 8
Cas 4 : 0 a 1 b	Cas 8 : 2 3 4 5 6 7	Cas 12 : 0 4 8 12 16

Pour le cas 10, on aurait pu écrire simplement

```
for n in range(0,10,4): # ou bien range(0,9,4)
    print(n)
```

Bien noter la différence entre le cas 11 et le cas 12 : dans ce dernier la partie correspondante à la condition `n%4==0` n'est jamais prise en compte car la condition `n%2==0` étant forcement vérifiée pour les mêmes cas, elle sera traitée avant. En règle générale, lors de l'écriture de conditions, toujours commencer par la condition la plus restrictive.

Exercice 4.2 (Mes ingrédients préférés)

Créez une liste contenant cinq ingrédients indispensables pour une bonne soupe, par exemple :

```
ingredients = [ "escargots", "sangsues", "tranches de gorilles", "sourcils de chenilles"]
```

Ensuite, créez une boucle qui affiche le contenu de cette liste avec des numéros comme suit :

```
1 escargots
2 sangsues
3 tranches de gorilles
4 sourcils de chenilles
```

Correction

```
ingredients = [ "escargots",                    1 escargots
                "sangsues",                     2 sangsues
                "tranches de gorilles",         3 tranches de gorilles
                "sourcils de chenilles"]        4 sourcils de chenilles

for idx, item in enumerate(ingredients):
    print(idx + 1, item)
```

Exercice 4.3 (Devine le résultat)

Quel résultat donne le code suivant ? Après avoir écrit la réponse, vérifier avec l'ordinateur.

```
L_1 = list(range(0,11,2))
L_2 = list(range(1,12,2))
L = []
for i in range(len(L_1)):
    L = L + [L_1[i]] + [L_2[i]]
    # idem que L.append(L_1[i]).append(L_2[i])
    # ou encore L.extend([ L_1[i],L_2[i] ])
print(L_1, "\t", L_2, "\t", L)
```

Correction

`[0, 2, 4, 6, 8, 10]` → `[1, 3, 5, 7, 9, 11]` → `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]`

Exercice Bonus 4.4 (Devine le résultat – modification de listes)

Quel résultat donne chacun des codes suivants ? Écrire la réponse, puis vérifier en les exécutant.

```
liste = [1, 2, 3]
for item in liste:
    item = 0
print(liste)
```

```
liste = [1, 2, 3]
for idx in range(3):
    liste [idx] = 0
print(liste)
```

```
liste = [[1], [2], [3]]
for item in liste:
    item[0] = 0
print(liste)
```

Correction

`[1, 2, 3]` `[0, 0, 0]` `[[0], [0], [0]]`

Dans le premier code, la variable `item` reçoit la valeur de chaque élément, mais la liste n'est pas modifiée. On peut voir cela comme l'équivalent de :

```
for idx in range(3):
    item = liste [idx] # on copie la valeur contenue dans la case liste[idx] dans la boîte item
    item = ... # liste[idx] inchangée
```

Avec le deuxième code, on modifie directement la liste via l'affectation `liste[idx] = 0`.

Dans le troisième code (listes dans une liste), chaque `item` est une référence vers une sous-liste. Ainsi `item[0] = 0` modifie directement cette sous-liste. On peut voir cela comme l'équivalent de :

```
for idx in range(3):
    item = liste[idx] # on copie l'adresse de la sous-liste liste[idx] dans la boîte item
    item = ... # la sous-liste liste[idx] est modifiée
```

En résumé,

- pour modifier une liste, on utilise une affectation directe : `liste[idx] = ...` ;
- `for item in liste` crée une variable `item` qui est une **copie de la valeur** (si immuable) ou une **référence** (si mutable)

Pour aller plus loin : https://gervais.forge.apps.education.fr/tnsi/00_Distri/Variables_en_Python.pdf

Attention

L'exercice 4.5 est le premier d'une longue liste d'exercices tirés des pydéfis, dont la liste complète est ici https://pydefis.callicode.fr/user/liste_defis. Il s'agit des énigmes des concours annuels c0d1ngUp http://codingup.fr/, un challenge de programmation, en temps limité et en présentiel, soutenu par l'Université de Poitiers. Durant une journée les participants font face à une série de problèmes qu'ils doivent résoudre par le moyen de leur choix. Les problèmes sont choisis pour que leur résolution nécessite de la programmation, de l'inventivité et de l'ingéniosité.

L'auteur aimerait que le site puisse continuer à être utilisé par des enseignants, avec leurs élèves. C'est impossible si les solutions toutes prêtes fleurissent. C'est pour cette raison que je ne publie pas les solutions même dans la version corrigée de ce polycopié. Pour vérifier vos réponse, vous pouvez me demander ou, encore mieux, vérifier directement sur le site des Pydéfis.

Exercice Bonus 4.5 (Pydéfis – L'algorithme du professeur Guique)

Pour dissimuler la combinaison à trois nombres de son coffre, le professeur Guique a eu l'idée de la cacher à l'intérieur d'un algorithme. Les trois nombres de la combinaison sont en effet les valeurs contenues dans les variables a, b et c après l'exécution de l'algorithme suivant :

```
Initialiser a, b, c, k et n respectivement a 1, 2, 5, 1 et 0
Répéter tant que k est strictement inférieur a 1000-n
    a = b
    b = c + a
    c = 3c + 4a - b
    n = a + b
    augmenter k de 1
fin répéter
Afficher a, b, c
```

Quelle est la séquence des trois nombres ouvrant le coffre du professeur Guique ?

Source (avec des valeurs initiales différentes) : https://pydefis.callicode.fr/defis/Algorithme/txt

Exercice 4.6 (Pièces magiques)

En creusant dans le fond du jardin, on découvre un vieux sac contenant 20 pièces d'or. Le lendemain, la curiosité nous pousse à les placer dans la machine à dupliquer de notre grand-père inventeur. Miracle : chaque jour, la machine ajoute 10 pièces supplémentaires. Mais un corbeau chapardeur découvre notre cachette et revient chaque semaine voler 3 pièces.

On souhaite suivre l'évolution du trésor pendant un an. Créer la liste `S` contenant le nombre total de pièces en fin de chaque semaine : `S[0]=20`, `S[1]=20+10*7-3`, etc. (pour les 52 semaines de l'année).

Correction

On part de 20 pièces ; chaque semaine on en gagne 10×7 grâce à la machine et on en perd 3 à cause du corbeau. Posons S_n le nombre total de pièces à la fin de la semaine n. On a $S_0 = 20$ et, chaque semaine, on gagne $10 \times 7 = 70$ pièces magiques et on en perd 3 à cause du corbeau, soit un gain net de $70 - 3 = 67$ pièces par semaine. Ainsi, la suite (S_n) est définie par la récurrence

$$\begin{cases} S_0 = 20, \\ S_{n+1} = S_n + 67. \end{cases}$$

```
found, weekly_gain, weekly_stolen = 20, 10*7, 3

S = [found]
for week in range(52):  # 52 semaines dans l'année
    S.append( S[-1] + weekly_gain - weekly_stolen )

print(f"S[0]={S[0]}, S[1]={S[1]}, S[52]={S[52]}")
```

```
S[0]=20, S[1]=87, S[52]=3504
```

En comparaison, un calcul direct sur 365 jours donne :

```
>>> 20 + 10*365 - 3*52
3514
```

Notre programme trouve légèrement moins, car une année compte 52 semaines (364 jours), il manque donc le gain d'un jour, soit 10 pièces.

Exercice 4.7 (Lower to Upper)

Considérons la liste d'acronymes

```
acronymes = ["asap", "faq", "fyi", "diy"]
```

Tous les acronymes sont en minuscules. Avec une boucle `for` les mettre en majuscules (en les remplaçant dans la même liste).

Correction

```
acronymes = ["asap", "faq", "fyi", "diy"]
print(acronymes)
for i in range(len(acronymes)):
    acronymes[i] = acronymes[i].upper()
print(acronymes)
```

Output:

```
['asap', 'faq', 'fyi', 'diy']
['ASAP', 'FAQ', 'FYI', 'DIY']
```

Exercice 4.8 (Poids sur la Lune)

Si on se trouvait actuellement sur la Lune, notre poids ne représenterait que 16,5% de celui que l'on a sur la Terre. Pour calculer cela, il suffit de multiplier notre poids sur Terre par 0.165. Imaginons qu'on prenne un kilo de plus chaque année pendant les 15 prochaines années. Utilisons une boucle `for` pour afficher notre poids sur la Lune la première, cinquième, dixième et quinzième année.

Correction

```
kg_sur_terre = 70 # Poids sur Terre en kilogrammes
facteur_lune = 0.165 # Facteur pour le poids sur la Lune

for annee in [1,5,10,15]:
    kg_sur_terre += 1
    kg_sur_lune = kg_sur_terre * facteur_lune
    print(f"Après {annee} an(s), poids sur Terre = {kg_sur_terre} kg, sur Lune = {kg_sur_lune:.2f} kg")
```

```
Après 1 an(s), poids sur Terre = 71 kg, sur Lune = 11.71 kg
Après 5 an(s), poids sur Terre = 72 kg, sur Lune = 11.88 kg
Après 10 an(s), poids sur Terre = 73 kg, sur Lune = 12.04 kg
Après 15 an(s), poids sur Terre = 74 kg, sur Lune = 12.21 kg
```

Exercice 4.9 (Vente de voitures)

Vous travaillez pour un célèbre constructeur automobile et vous devez expédier les nouvelles voitures qui viennent d'arriver. Vos collègues ont indiqué les voitures destinées à la France, à l'Espagne et au Portugal, mais ils les ont mélangées:

```
cars = ["PT-754J", "ES-096L", "FR-245F", "PT-536G", "FR-543H", "PT-653H"]
```

À l'aide d'une boucle `for`, dispatchez les trois groupes de voitures dans trois listes en fonction de leurs destinations.

Correction

```
PT, ES, FR = [], [], []
for c in cars:
    if c[:2]=='PT': PT.append(c) # ok car une seule instruction
    elif c[:2]=='ES': ES.append(c)
    elif c[:2]=='FR': FR.append(c)
print(PT, ES, FR)
```

```
['PT-754J', 'PT-536G', 'PT-653H'] ['ES-096L'] ['FR-245F', 'FR-543H']
```

Mieux: un dictionnaire

```python
resultats = {"PT": [], "ES": [], "FR": []}
for c in cars:
    prefixe = c[:2]
    if prefixe in resultats:
        resultats[prefixe].append(c)
print(resultats)
```

```
{'PT': ['PT-754J', 'PT-536G', 'PT-653H'], 'ES': ['ES-096L'], 'FR': ['FR-245F', 'FR-543H']}
```

Exercice 4.10 (Multiplication sans "*")

On souhaite calculer le produit de deux nombres entiers n_1 et n_2. La seule opération autorisée est l'addition.

Correction

Pour effectuer le produit on distingue deux cas de figures: si $n_1 \geq 0$, on se contente d'additionner n_1 fois le nombre n_2; si $n_1 < 0$, on échange le signe des deux facteurs.

```python
n1, n2 = 10, -23

if n1 < 0: n1, n2 = -n1, -n2

p = 0
for i in range(1,n1+1): p += n2

print(f"({n1}) x ({n2}) = {p}")
```

```
(10) x (-23) = -230
```

Exercice 4.11 (Triangles)

Créer des scripts qui dessinent des triangles comme les suivants:

```
x          xxxxx          x        xxxxx
xx         xxxx          xx         xxxx
xxx        xxx          xxx          xxx
xxxx       xx          xxxx           xx
xxxxx      x          xxxxx            x
```

Correction

```python
for n in range(1, 6):
    print('x' * n)

for n in range(5, 0, -1):
    print('x' * n)
```

ou

```python
print('x\nxx\nxxx\nxxxx\nxxxxx')
print('xxxxx\nxxxx\nxxx\nxx\nx')
print('    x\n   xx\n  xxx\n xxxx\nxxxxx')
print('xxxxx\n xxxx\n  xxx\n   xx\    x')
```

```python
for n in range(1, 6):
    print(' ' * (5 - n) + 'x' * n)

for n in range(5, 0, -1):
    print(' ' * (5 - n) + 'x' * n)
```

Exercice 4.12 (Nombre de voyelles)

Pour un texte donné (sans accent), créer un programme qui affiche le nombre de voyelles et de consonnes.

Correction

Analyse du problème. Pour résoudre cet exercice, nous devons déterminer le nombre de voyelles et de consonnes dans un texte donné. Les voyelles sont les lettres "a", "e", "i", "o", "u" et "y". Les consonnes sont toutes les autres lettres de l'alphabet sauf les espaces et la ponctuation. Pour faire simple, nous considérons uniquement les lettres de l'alphabet et la ponctuation et ne prenons pas en compte les chiffres ou les caractères spéciaux.

Algorithme. On initialise les compteurs de voyelles et de consonnes à zéro. On parcoure ensuite chaque caractère du texte. Si le caractère est une voyelle, on augmente le compteur de voyelles de 1. Sinon, on augmente le compteur de consonnes de 1.

Implémentation. Voici une implémentation de cet algorithme avec application sur un exemple :

```
texte = "Exemple de texte pour compter les voyelles et consonnes."
v, c = 0, 0
for lettre in texte.lower():
    if lettre in "aeiouy":
        v += 1
    elif lettre not in " ,:;.!?":
        c += 1

print(f"Voyelles = {v}, Consonnes = {c}")
```

```
Voyelles = 19, Consonnes = 28
```

Exercice 4.13 (Cartes de jeux)

On considère les deux listes suivantes :

- `couleurs = ['pique','coeur','carreau','trefle']`
- `valeurs = ['7','8','9','10','valet','reine','roi','as']`

Écrire un script qui affiche toutes les cartes d'un jeu de 32 cartes.

Correction

```
couleurs = ['pique','coeur','carreau','trefle']
valeurs = ['7','8','9','10','valet','reine','roi','as']
for c in couleurs:
    for v in valeurs:
        print(v, c)
```

7 pique	8 pique	9 pique	10 pique	valet pique	reine pique	roi pique	as pique
7 coeur	8 coeur	9 coeur	10 coeur	valet coeur	reine coeur	roi coeur	as coeur
7 carreau	8 carreau	9 carreau	10 carreau	valet carreau	reine carreau	roi carreau	as carreau
7 trefle	8 trefle	9 trefle	10 trefle	valet trefle	reine trefle	roi trefle	as trefle

Exercice 4.14 (Tables de multiplication)

Afficher les tables de multiplication entre 1 et 9. Voici un exemple de ligne à afficher : `7 x 9 = 63`.

On verra à l'exercice 5.31 une autre présentation des tables de multiplication sous la forme du tableau de Pythagore.

Correction

```
for b in range(1,10):
    for a in range(1,10):
        print(f'{a} x {b} = {a*b}')
    print('')
```

1 x 1 = 1	1 x 2 = 2	1 x 3 = 3	1 x 4 = 4	1 x 5 = 5	1 x 6 = 6	1 x 7 = 7	1 x 8 = 8	1 x 9 = 9
2 x 1 = 2	2 x 2 = 4	2 x 3 = 6	2 x 4 = 8	2 x 5 = 10	2 x 6 = 12	2 x 7 = 14	2 x 8 = 16	2 x 9 = 18
3 x 1 = 3	3 x 2 = 6	3 x 3 = 9	3 x 4 = 12	3 x 5 = 15	3 x 6 = 18	3 x 7 = 21	3 x 8 = 24	3 x 9 = 27
4 x 1 = 4	4 x 2 = 8	4 x 3 = 12	4 x 4 = 16	4 x 5 = 20	4 x 6 = 24	4 x 7 = 28	4 x 8 = 32	4 x 9 = 36
5 x 1 = 5	5 x 2 = 10	5 x 3 = 15	5 x 4 = 20	5 x 5 = 25	5 x 6 = 30	5 x 7 = 35	5 x 8 = 40	5 x 9 = 45
6 x 1 = 6	6 x 2 = 12	6 x 3 = 18	6 x 4 = 24	6 x 5 = 30	6 x 6 = 36	6 x 7 = 42	6 x 8 = 48	6 x 9 = 54
7 x 1 = 7	7 x 2 = 14	7 x 3 = 21	7 x 4 = 28	7 x 5 = 35	7 x 6 = 42	7 x 7 = 49	7 x 8 = 56	7 x 9 = 63
8 x 1 = 8	8 x 2 = 16	8 x 3 = 24	8 x 4 = 32	8 x 5 = 40	8 x 6 = 48	8 x 7 = 56	8 x 8 = 64	8 x 9 = 72
9 x 1 = 9	9 x 2 = 18	9 x 3 = 27	9 x 4 = 36	9 x 5 = 45	9 x 6 = 54	9 x 7 = 63	9 x 8 = 72	9 x 9 = 81

Exercice Bonus 4.15 (Tables de vérité)

La méthode des tables de vérité est une méthode élémentaire pour tester la validité d'une formule du calcul propositionnel. Les énoncés étant composés à partir des connecteurs «non», «et», «ou», «si..., alors», «si et seulement si», notés respectivement $\neg$, $\wedge$, $\vee$, $\Longrightarrow$, $\Longleftrightarrow$, les fonctions de vérités du calcul propositionnel classique sont données par la table de vérité suivante:

P	Q	$\neg$P	$\neg$Q	P $\wedge$ Q	P $\vee$ Q	P $\Longrightarrow$ Q	Q $\Longrightarrow$ P	P $\Longleftrightarrow$ Q
F	F	V	V	F	F	V	V	V
F	V	V	F	F	V	V	F	F
V	F	F	V	F	V	F	V	F
V	V	F	F	V	V	V	V	V

Générer les valeurs de cette table automatiquement avec python *(on ne demande pas d'afficher le résultat sous forme de tableau).*

Correction

```
print(f'{"P":<8}{"Q":<8}{"non P":<8}{"non Q":<8}{"P et Q":<8}{"P ou Q":<8}{"P => Q":<8}{"Q => P":<8}{"P
↪  ssi Q":<8}')
for P in [False,True]:
    for Q in [False,True]:
        print(f'{P:<8}{Q:<8}{not P:<8}{not Q:<8}{P and Q:<8}{P or Q:<8}{(not P) or Q:<8}{(not Q) or
        ↪  P:<8}{((not P) or Q) and ((not P) or Q):<8}')
```

```
P       Q       non P   non Q   P et Q  P ou Q  P => Q  Q => P  P ssi Q
0       0       1       1       0       0       1       1       1
0       1       1       0       0       1       1       0       1
1       0       0       1       0       1       0       1       0
1       1       0       0       1       1       1       1       1
```

Notons un comportement particulier dans le formatage f-string: les valeurs booléennes sont interprétées comme des entiers, où `True` est équivalent à 1 et `False` à 0. Pour afficher les booléens, on peut forcer la conversion:

```
print(f'{"P":<8}{"Q":<8}{"non P":<8}{"non Q":<8}{"P et Q":<8}{"P ou Q":<8}{"P => Q":<8}{"Q => P":<8}{"P
↪  ssi Q":<8}')
for P in [False,True]:
    for Q in [False,True]:
        print(f'{str(P):<8}{str(Q):<8}{str(not P):<8}{str(not Q):<8}{str(P and Q):<8}{str(P or
        ↪  Q):<8}{str((not P) or Q):<8}{str((not Q) or P):<8}{str(((not P) or Q) and ((not P) or
        ↪  Q)):<8}')
```

```
P       Q       non P   non Q   P et Q  P ou Q  P => Q  Q => P  P ssi Q
False   False   True    True    False   False   True    True    True
False   True    True    False   False   True    True    False   True
True    False   False   True    False   True    False   True    False
True    True    False   False   True    True    True    True    True
```

Bonus: lorsqu'on aura introduit les modules, on pourra utiliser le module `tabulate` qui prend en entrée une liste:

```
from tabulate import tabulate
L = []
L.append(['P', 'Q', 'non P', 'non Q', 'P et Q', 'P ou Q', 'P => Q', 'Q => P', 'P ssi Q'])
for P in [False,True]:
    for Q in [False,True]:
        L.append([P,Q,not P,not Q,P and Q,P or Q,(not P) or Q,(not Q) or P,((not P) or Q) and ((not P)
        ↪  or Q)])

print(tabulate(L, headers='firstrow'))
```

```
P      Q      non P    non Q    P et Q    P ou Q    P => Q    Q => P    P ssi Q
-----  -----  -------  -------  --------  --------  --------  --------  ---------
False  False  True     True     False     False     True      True      True
False  True   True     False    False     True      True      False     True
True   False  False    True     False     True      False     True      False
True   True   False    False    True      True      True      True      True
```

Exercice 4.16 (Parcourir deux listes)

On considère les deux listes suivantes (les deux listes ont la même taille) :

```
a = [8468, 4560, 3941, 3328, 7, 9910, 9208, 8400, 6502, 1076, 5921, 6720, 948, 9561, 7391, 7745,
↪  9007, 9707, 4370, 9636, 5265, 2638, 8919, 7814, 5142, 1060, 6971, 4065, 4629, 4490, 2480, 9180,
↪  5623, 6600, 1764, 9846, 7605, 8271, 4681, 2818, 832, 5280, 3170, 8965, 4332, 3198, 9454, 2025]

b = [9093, 2559, 9664, 8075, 4525, 5847, 67, 8932, 5049, 5241, 5886, 1393, 9413, 8872, 2560, 4636,
↪  9004, 7586, 1461, 350, 2627, 2187, 7778, 8933, 351, 7097, 356, 4110, 1393, 4864, 1088, 3904,
↪  5623, 8040, 7273, 1114, 4394, 4108, 7123, 8001, 5715, 7215, 7460, 5829, 9513, 1256, 4052, 1585]
```

Trouvez le(s) nombre(s) qui sont exactement à la même place dans la liste `a` et dans la liste `b`.

Correction

On peut parcourir deux listes de deux façons :

```python
# Parcours simultané avec zip
for item_L, item_M in zip(L, M):
    item_L, item_M
```

```python
# Parcours par indices
for idx in range(min(len(L), len(M))):
    L[idx], M[idx]
```

Si l'on utilise `zip`, on peut combiner avec `enumerate` pour récupérer la position :

```python
for i, (ai, bi) in enumerate(zip(a, b)):
    if ai == bi:
        print(f"En position {i} : a[{i}] = {ai}, b[{i}] = {bi}")
```

```
En position 32 : a[32] = 5623, b[32] = 5623
```

On peut aussi parcourir les listes par indices, méthode traditionnelle :

```python
for i in range(len(a)):
    if a[i] == b[i]:
        print(f"En position {i} on a a[{i}] = {a[i]} et b[{i}] = {b[i]}.")
```

```
En position 32 on a a[32] = 5623 et b[32] = 5623.
```

La méthode avec `zip` est généralement plus lisible et évite de manipuler les indices directement.

Exercice 4.17 (Parcourir deux chaînes de caractères)

On considère des mots à trous : ce sont des chaînes de caractères contenant uniquement des majuscules et des caractères "`*`". Par exemple "`INFO*MA*IQUE`", "`***I***E**`" et "`*S*`" sont des mots à trous.

Pour deux chaînes de caractères `mot` et `mot_a_trous` données, afficher `True` si on peut obtenir `mot` en remplaçant convenablement les caractères "`*`" de `mot_a_trous`, `False` sinon.

Correction

Sujet 38.1 : https://glassus.github.io/terminale_nsi/T6_6_Epreuve_pratique/BNS_2023/?s=03 ou encore https://codex.forge.apps.education.fr/exercices/correspond_mot/

On peut parcourir deux chaînes de caractères simultanément soit par indices, soit à l'aide de la fonction `zip`. Voici une solution utilisant `zip`, plus lisible et naturelle :

```python
mot, mot_a_trou = "INFORMATIQUE", "INFO*MA*IQUE"    # True
# mot, mot_a_trou = "AUTOMATIQUE", "INFO*MA*IQUE"  # False
# mot, mot_a_trou = "BOL", "V*L"                   # False
# mot, mot_a_trou = "AUTO", "*UT*"                 # True

reponse = True
if len(mot) != len(mot_a_trou):
    reponse = False
else:
    for c, t in zip(mot, mot_a_trou):
        if t != '*' and c != t:
            reponse = False
            break
print(reponse)
```

```
True
```

On peut aussi écrire une version équivalente en utilisant les indices directement :

```
reponse = True
if len(mot) != len(mot_a_trou):
    reponse = False
else:
    for i in range(len(mot)):
        if mot_a_trou[i] != '*' and mot_a_trou[i] != mot[i]:
            reponse = False
            break
print(reponse)
```

```
True
```

Exercice 4.18 (Devine le résultat - `while`)

Quel résultat donnent les codes suivants ? Après avoir écrit votre réponse, vérifiez-là avec l'ordinateur.

Cas 1 :
```
n = 1
while n<5:
    print(n)
    n += 1
```

Cas 2 :
```
n = 1
while n<5:
    n += 1
    print(n)
```

Cas 3 :
```
n = 1
while (n<10) and (n<5):
    print(n)
    n += 1
```

Cas 4 :
```
n = 1
while (n<10) or (n<5):
    print(n)
    n += 1
```

Cas 5 :
```
n = 1
while 5<n<10:
    print(n)
    n += 1
```

Cas 6 :
```
n = 6
while 5<n<10:
    print(n)
    n += 1
```

Cas 7 :
```
n = 9
while 5<n<10:
    print(n)
    n -= 1
```

Cas 8 :
```
i = 0
n = 0
while n<10:
    print(i,n)
    i += 1
    n = i*i
```

Cas 9 :
```
i = 0
n = 0
while n<10:
    print(i,n)
    n = i*i
    i += 1
```

Correction

Notons que "`(n<10) and (n<5)`" équivaut à "`(n<5)`" tandis que "`(n<10) or (n<5)`" équivaut à "`(n<10)`".

Cas 1 : 1 2 3 4

Cas 2 : 2 3 4 5

Cas 3 : 1 2 3 4

Cas 4 : 1 2 3 4 5 6 7 8 9

Cas 5 :

Cas 6 : 6 7 8 9

Cas 7 : 9 8 7 6

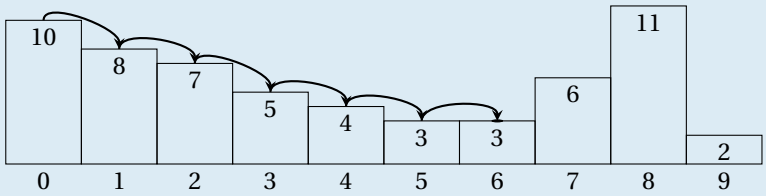
Cas 8 : 0 0 1 1 2 4 3 9

Cas 9 : 0 0 1 0 2 1 3 4 4 9

Exercice 4.19 (CodEx : Premier minimum local)

Une balle roule sur un chemin dallé où certaines dalles sont plus basses et d'autres plus hautes. Les hauteurs des dalles sont données sous forme d'une liste d'entiers positifs (elle contient au moins deux valeurs). Pour une liste `hauteurs` donnée, déterminer l'indice de la dalle sur laquelle s'arrête la balle.

Dans l'exemple ci-dessous, la balle s'arrête sur la dalle d'indice 6, car elle se pose sur la première dalle dont la hauteur est strictement inférieure à celle de la suivante. Lorsque deux dalles consécutives sont à la même hauteur, la balle continue de rouler.



Source : https://codex.forge.apps.education.fr/exercices/premier_minimum/

Correction
Pour déterminer l'indice de la dalle sur laquelle s'arrête la balle, on commence par initialiser l'indice i à 0 ; puis parcourons les indices à partir de 0 tant que la hauteur de la dalle suivante est inférieure ou égale à celle de la dalle actuelle (*i.e.* tant que hauteurs$[i] \geq$ hauteurs$[i+1]$) et incrémentons i à chaque itération. Lorsque la condition est fausse, i est l'indice de la dalle où la balle s'arrête. Il s'agit d'une recherche de minimum par une approche gloutonne.

```
hauteurs = [10, 8, 7, 5, 4, 3, 3, 6, 11, 2]
i = 0
while hauteurs[i] >= hauteurs[i + 1]:
    i += 1
print(f"La balle s'arrête sur la dalle d'indice {i}.")
```

```
La balle s'arrête sur la dalle d'indice 6.
```

Exercice 4.20 (Suites type ①: $u_n = f(n)$. Recherche de seuil: plus petit n tel que...)
Soit une suite de la forme $u_n = f(n)$. On souhaite déterminer le plus petit indice n tel que $u_n \geq$ seuil , c'est-à-dire n tel que $u_{n-1} <$ seuil $\leq u_n$.

Écrire un script qui affiche le plus petit entier n lorsque seuil $=10\,000$ dans les trois cas suivants :

a) $u_n = (n+1)(n+3)$, b) $u_n = 4+5+6+\cdots+n$, c) $u_n = 1^2+2^2+3^2+\cdots+n^2$.

Pour résoudre ces problèmes, il est nécessaire d'utiliser une boucle `while` puisque le nombre d'itérations n'est pas connu à l'avance. Assurez-vous d'initialiser et de mettre à jour correctement le compteur dans la boucle. La boucle doit continuer **tant que le critère n'est pas satisfait** ; par exemple, si on cherche le plus petit n tel que $f(n) > s$, il faudra répéter les calculs tant que $f(n) \leq s$ car la négation de $>$ est $\leq$ etc.

Correction
Nous traitons ici des suites de la forme $u_n = f(n)$ et nous cherchons le plus petit entier n tel que $u_n \geq s$ avec $s = 10\,000$, c'est-à-dire n tel que $u_{n-1} < s \leq u_n$.

1. $\boxed{u_n = (n+1)(n+3)}$

```
n = 0
while (n+1)*(n+3) < 10**4: # c'est-à-dire    not( (n+1)*(n+3) >= 10**4 )
    n += 1
print(f"n = {n}")
# VALIDATION
print(f" • si n = {n} alors u_n = {(n+1)*(n+3)}")
m = n-1
print(f" • si n = {m} alors u_n = {(m+1)*(m+3)}")
```

```
n = 99
 • si n = 99 alors u_n = 10200
 • si n = 98 alors u_n = 9999
```

Dans ce cas, on peut calculer $n \in \mathbb{N}$ analytiquement :

$$u_{n-1} < 10\,000 \leq u_n \iff n(n+2) < 10^4 \leq (n+1)(n+3) \iff \begin{cases} n(n+2) - 10^4 < 0 \\ (n+1)(n+3) - 10^4 \geq 0 \end{cases}$$

$$\iff \begin{cases} n < -1 + \sqrt{1+10000} \approx 99.005 \\ n \geq -2 + \sqrt{4+9997} \approx 98.005 \end{cases} \iff n = 99.$$

Remarque : une fois le chapitre sur les fonctions abordé (chapitre 6), nous pourrons encapsuler le calcul de u_n dans une fonction :

```
u = lambda x: (x+1)*(x+3)
n = 0
seuil = 10000
while not( u(n)>=seuil ) :
    n += 1
print(f"u({n})={u(n)} et u({n-1})={u(n-1)}")
```

```
u(99)=10200 et u(98)=9999
```

2. $\boxed{u_n = \sum_{i=4}^{n} i}$

```
n = 4
somme = n
while somme < 10**4 :
    n += 1
    somme += n
print(f"n = {n}")
print(f" • si n = {n} alors somme = {somme}")
print(f" • si n = {n-1} alors somme = {somme-n}")
```

```
n = 141
 • si n = 141 alors somme = 10005
 • si n = 140 alors somme = 9864
```

Dans ce cas aussi on peut calculer $n \in \mathbb{N}$ analytiquement car $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$:

$$\sum_{i=4}^{n-1} i \leq 10^4 < \sum_{i=4}^{n} i \iff 0 \leq 10^4 - \sum_{i=4}^{n-1} i < n \iff 0 \leq 10^4 - \left(\frac{(n-1)n}{2} - 1 - 2 - 3\right) < n \iff 0 \leq 10^4 + 6 - \frac{(n-1)n}{2} < n$$

$$\iff \begin{cases} n^2 + n - 2 \times 10006 > 0, \\ n^2 - n - 2 \times 10006 \leq 0 \end{cases} \iff \begin{cases} n > \frac{-1+\sqrt{1+8\times 10006}}{2} = \frac{-1+\sqrt{80049}}{2} \approx 140.9647, \\ n \leq \frac{1+\sqrt{1+8\times 10006}}{2} = \frac{1+\sqrt{80049}}{2} \approx 141.9647. \end{cases}$$

3. $\boxed{u_n = \sum_{i=1}^{n} i^2}$

```
n = 1
somme = n**2
while somme < 10**4 :
    n += 1
    somme += n**2
print(f"n = {n}")
print(f" • si n = {n} alors somme = {somme}")
print(f" • si n = {n-1} alors somme = {somme-n**2}")
```

```
n = 31
 • si n = 31 alors somme = 10416
 • si n = 30 alors somme = 9455
```

Bien que nous puissions toujours théoriquement déterminer $n \in \mathbb{N}$ analytiquement car $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$, cette fois-ci nous devons calculer les racines d'un polynôme de degré 3. Les formules associées existent mais sont en général bien plus complexes que celles pour un polynôme de degré 2.

Exercice 4.21 (Suites type ①: $u_n = f(n)$ (suite géométrique))

Soit $(u_n)_{n\in\mathbb{N}}$ la suite définie par $u_n = (0.7)^{3n}$. On peut montrer que cette suite tend vers 0 lorsque n tend vers $+\infty$. Cela signifie que

$$\forall \varepsilon > 0 \quad \exists n \in \mathbb{N} \quad : \quad \forall n \geq \mathrm{N} \quad |u_n| < \varepsilon.$$

Déterminer le plus petit entier n tel que l'on ait $|u_n| < \varepsilon$ avec $\varepsilon = 10^{-4}$.

Indice : vérifiez par exemple que $u_7 \approx 0.00056$.

Correction

Nous pouvons calculer n par un calcul itératif :

```
n = 0
while (0.7)**(3*n) >= 1e-4:
    n += 1
print(n)
# VERIFICATIONS
print((0.7)**(3*n) < 1e-4) # u_n < ε
print(1e-4 <= (0.7)**(3*(n-1))) # u_{n-1} ≥ ε
```

```
9
True
True
```

On peut aussi procéder analytiquement: $u_n = \left(\frac{7}{10}\right)^{3n}$. Il s'agit d'une suite géométrique de raison $0 < q < 1$, donc décroissante avec $\lim_n u_n = 0$. On a

$$u_n < 10^{-4} \iff \left(\frac{7}{10}\right)^{3n} < 10^{-4} \iff \log_{10}\left(\frac{7}{10}\right)^{3n} < -4 \iff 3n\log_{10}\left(\frac{7}{10}\right) < -4 \overset{\log_{10}\left(\frac{7}{10}\right)<0}{\iff} n > -\frac{4}{3\log_{10}\left(\frac{7}{10}\right)} \simeq 8.6.$$

La plus petite valeur entière vérifiant cette inégalité est donc $n = 9$.

Remarque: une fois le chapitre sur les fonctions abordé (chapitre 6), nous pourrons réécrire notre code de manière plus modulaire, par exemple en encapsulant le calcul de u_n dans une fonction. Ainsi, au lieu de réécrire l'expression à chaque étape, nous utiliserons une fonction comme `f = lambda n: (0.7)**(3*n)`, ce qui rend le code plus lisible. De plus, pour mieux visualiser la décroissance, affichons tous les termes u_i pour $i = 0, \dots, n$:

```
f = lambda n: (0.7)**(3*n)
n = 0
u = f(n)
print(f"u_{n}={u:1.5f}")
while u >= 1.e-4:
    n += 1
    u = f(n)
    print(f"u_{n}={u:1.5f}")
```

```
u_0=1.00000
u_1=0.34300
u_2=0.11765
u_3=0.04035
u_4=0.01384
u_5=0.00475
u_6=0.00163
u_7=0.00056
u_8=0.00019
u_9=0.00007
```

Exercice 4.22 (Recherche de seuil: plus petit n tel que...)

Soit $n \in \mathbb{N}$ donné et cherchons la première puissance de 2 plus grande que n en utilisant une boucle `while`. Autrement dit, on cherche 2^p avec p tel que $2^{p-1} \leq n < 2^p$.

Correction

Méthode 1:

```
n = 100
p = 0
while 2**p<=n:
    p += 1 # p = p+1
print(2**p)
```

```
128
```

Méthode 2:

```
n = 100
p2 = 1
while p2<=n:
    p2 *= 2 # p2 = p2*2
print(p2)
```

```
128
```

Bonus: $2^p = q \iff p = \log_2 q$

```
from math import log
n = 100
p = int(log(n,2))+1
print(2**p)
```

```
128
```

Exercice 4.23 (Recherche de seuil: méthode du rectangle à gauche)

Soit $f\colon [a;b] \to \mathbb{R}$ une fonction intégrable et notons $\mathrm{I} = \int_a^b f(x)\,\mathrm{d}x$. Une valeur approchée de cette intégrale peut être obtenue par la méthode composite du rectangle à gauche:

$$\mathrm{J}(n) = \frac{b-a}{n}\sum_{i=0}^{n-1} f\left(a + i\frac{b-a}{n}\right).$$

On cherche le plus petit n tel que $|\mathrm{I} - \mathrm{J}(n)| < \varepsilon$ pour une tolérance ε, une fonction f et un intervalle $[a;b]$ choisis. Par exemple, si $\varepsilon = 10^{-3}$, $f(x) = 3x^2$, $a = 0$ et $b = 1$, alors $\mathrm{I} = \int_0^1 3x^2\,\mathrm{d}x = [x^3]_0^1 = 1$ et le plus petit n tel que $|\mathrm{I} - \mathrm{J}(n)| < \varepsilon$ est 1500.

Correction

Dans le code ci-dessous, on utilise une `fonction lambda` pour définir rapidement $f(x)$. Cette notation sera expliquée plus en détail au chapitre 6. Vous pouvez la remplacer par la formule directement.

```
f = lambda x : 3*x**2 # f: x↦3x², voir le chapitre 6
a, b, I, toll = 0, 1, 1, 1.e-3 # Bornes, Intégrale, Tolérance

# Définition de J(n) selon la méthode du rectangle à gauche
H = b - a
J = lambda n : H/n * sum([f(a + i*H/n) for i in range(n)])
```

```
# Recherche du plus petit n tel que |I - J(n)| < toll
n = 1
while abs(I - J(n)) >= toll:
    n += 1

print(f"{n = }, {J(n) = :g}, {I = }")
```

```
n = 1500, J(n) = 0.999, I = 1
```

Pour cet exemple particulier, il est possible de faire une vérification manuelle :

$$J(n) = \frac{1}{n}\sum_{i=0}^{n-1} 3\left(\frac{i}{n}\right)^2 = \frac{3}{n^3}\sum_{i=0}^{n-1} i^2 = \frac{3}{n^3}\frac{2(n-1)n(2n-1)}{6} = \frac{(n-1)(2n-1)}{2n^2} = \frac{2n^2-3n+1}{2n^2} = 1-\frac{3n-1}{2n^2}, \quad \rightsquigarrow \quad |I-J(n)| = \frac{3n-1}{2n^2}.$$

La condition $|I-J(n)| < \varepsilon$ équivaut à

$$3n-1 < 2n^2\varepsilon \quad \Longleftrightarrow \quad 2\varepsilon n^2 - 3n + 1 > 0 \quad \Longleftrightarrow \quad n > \frac{3+\sqrt{9-8\varepsilon}}{4\varepsilon} = \frac{3+\sqrt{9-0.008}}{0.004} \approx 1499.67$$

Exercice 4.24 (Suite type ② : récurrence $u_{n+1} = f(u_n)$)

Soit $(u_n)_{n\in\mathbb{N}}$ une suite définie par récurrence. Écrire un script qui affiche $u_0, \dots u_6$ en utilisant d'abord une boucle `while` puis une boucle `for` :

$$\begin{cases} u_0 = 1, \\ u_{n+1} = 2u_n + 1; \end{cases}$$

Ne pas oublier de vérifier vos calculs, par exemple on vérifiera que $u_3 = 15$.

Correction

	Boucle `while`	Boucle `for`	Output

Sans stockage	`n = 0`
	`u = 1`
	`print(f"u_{n} = {u}")`
	`while n < 6:`
	`    u = 2*u + 1`
	`    print(f"u_{n+1} = {u}")`
	`    n += 1`
		`u = 1`
		`print(f"u_{0} = {u}")`
		`for n in range(6):`
		`    u = 2*u + 1`
		`    print(f"u_{n+1} = {u}")`
			`u_0 = 1`
			`u_1 = 3`
			`u_2 = 7`
			`u_3 = 15`
			`u_4 = 31`
			`u_5 = 63`
			`u_6 = 127`

Avec append	`n = 0`
	`u = [1]`
	`while n < 6:`
	`    u.append(2*u[n] + 1)`
	`    n += 1`
	`for i,ui in enumerate(u):`
	`    print(f"u_{i} = {ui}")`
		`u = [1]`
		`for n in range(6):`
		`    u.append(2*u[n] + 1)`
		`for i,ui in enumerate(u):`
		`    print(f"u_{i} = {ui}")`
			`u_0 = 1`
			`u_1 = 3`
			`u_2 = 7`
			`u_3 = 15`
			`u_4 = 31`
			`u_5 = 63`
			`u_6 = 127`

Avec préallocation	`N = 6; u = [ None ]*(N+1)`
	`n = 0; u[n] = 1`
	`while n < N:`
	`    u[n+1] = 2*u[n] + 1`
	`    n += 1`
	`for i,ui in enumerate(u):`
	`    print(f"u_{i} = {ui}")`
		`N = 6; u = [ None ]*(N+1)`
		`u[0] = 1`
		`for n in range(N):`
		`    u[n+1] = 2*u[n] + 1`
		`for i,ui in enumerate(u):`
		`    print(f"u_{i} = {ui}")`
			`u_0 = 1`
			`u_1 = 3`
			`u_2 = 7`
			`u_3 = 15`
			`u_4 = 31`
			`u_5 = 63`
			`u_6 = 127`

Remarque : lorsqu'on aura vu le chapitre 6 sur les fonctions, on pourra par exemple écrire :

```
f = lambda a : 2*a+1 # f : a → 2a+1
u = [1]
for n in range(6):
    u.append(f(u[-1]))
for i,ui in enumerate(u):
    print(f"u_{i} = {ui}")
```

```
u_0 = 1        u_1 = 3        u_2 = 7        u_3 = 15       u_4 = 31       u_5 = 63       u_6 = 127
```

Exercice 4.25 (Suites type ②: $\sqrt{2\sqrt{2\sqrt{2\sqrt{2\ldots}}}}$)

On peut donner un sens au nombre b qui s'écrit

$$b = \sqrt{2\sqrt{2\sqrt{2\sqrt{2\ldots}}}}.$$

Lequel et que vaut b?

Correction

- b est la limite de la suite suivante définie par récurrence : $u_0 \in [1;2]$ et $u_{n+1} = \sqrt{2u_n}$. Nous arrêtons les calculs dès que la suite devient numériquement stationnaire :

```
uu = [ 1, 2**0.5  ] # u0 = 1 et u1 = √2
while uu[-1] != uu[-2] :
    uu.append( (2*uu[-1])**0.5 )
print("Nombre d'itérations :", len(uu), "Valeur :" , uu[-1])
```

```
Nombre d'itérations : 55 Valeur : 1.9999999999999998
```

- Montrons mathématiquement que la suite converge et calculons la valeur exacte de b.

  Suite bornée. Montrons par récurrence que $u_n \in [1;2]$ pour tout $n \in \mathbb{N}$.

  - Initialisation. On a $u_0 \in [1;2]$ par hypothèse.
  - Hérédité. Supposons que $u_k \in [1;2]$. Alors $2u_k \in [2;4]$, ce qui implique $u_{k+1} = \sqrt{2u_k} \in [\sqrt{2};\sqrt{4}] \subset [1;2]$.

  Par récurrence, on en déduit que $u_n \in [1;2]$ pour tout $n \in \mathbb{N}$.

  Monotonie. Pour tout $n \in \mathbb{N}$, on a

  $$\frac{u_{n+1}}{u_n} = \sqrt{\frac{2}{u_n}}.$$

  On a prouvé que $u_n \leq 2$ donc $\frac{2}{u_n} \geq 1$. Ainsi $u_{n+1} \geq u_n$, ce qui montre que la suite (u_n) est croissante.

  Convergence. La suite (u_n) est croissante et majorée par 2. Par le théorème de convergence des suites monotones, elle converge donc vers une limite b.

  Limite. Puisque $u_n \to b$, la relation de récurrence donne $b = \sqrt{2b}$. En élevant au carré, on obtient $b^2 = 2b$, soit encore $b(b-2) = 0$. Les solutions sont $b = 0$ ou $b = 2$. Or, $u_n > 1$ pour tout n, donc $b \geq 0$. Par conséquent, $b = 2$.

Exercice 4.26 (Suites type ②: $\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+\ldots}}}}$)

On peut donner un sens au nombre b qui s'écrit

$$b = \sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+\ldots}}}}.$$

Lequel et que vaut b?

Correction

- b est la limite de la suite suivante définie par récurrence : $u_0 = 0$ et $u_{n+1} = \sqrt{2+u_n}$. Nous arrêtons les calculs dès que la suite devient numériquement stationnaire :

```
uu = [0, 2**0.5] # u_0 = 1 et u_1 = √2
while uu[-1] != uu[-2] :
    uu.append( (2+uu[-1])**0.5 )
print("Nombre d'itérations :", len(uu), "Valeur :" , uu[-1])
```

```
Nombre d'itérations : 30 Valeur : 2.0
```

- Montrons mathématiquement que la suite converge et calculons la valeur exacte de b.

  Suite bornée. Montrons par récurrence que $u_n \in [\sqrt{2};2]$ pour tout $n \in \mathbb{N}_*$.

  - Initialisation. On a $u_1 = \sqrt{2}$.
  - Hérédité. Supposons que $u_k \in [\sqrt{2};2]$. Alors $2+u_k \in]2+\sqrt{2};4]$, ce qui implique $u_{k+1} = \sqrt{2+u_k} \in [\sqrt{2+\sqrt{2}};\sqrt{4}] \subset [\sqrt{2};2]$.

  Par récurrence, on en déduit que $u_n \in [\sqrt{2};2]$ pour tout $n \in \mathbb{N}_*$.

  Croissance. Pour tout $n \in \mathbb{N}$, on a

$$\left(\frac{u_{n+1}}{u_n}\right)^2 = \frac{2+u_n}{u_n^2}.$$

  Puisque $u_n \in [\sqrt{2};2]$ pour tout $n > 0$, on a

$$\frac{2+u_n}{u_n^2} = \frac{2}{u_n^2} + \frac{1}{u_n} \geq \frac{2}{2^2} + \frac{1}{2} = 1$$

  donc $u_{n+1} \geq u_n$, ce qui montre que la suite (u_n) est croissante.

  Convergence. La suite (u_n) est croissante et majorée par 2. Par le théorème de convergence des suites monotones, elle converge donc vers une limite b.

  Limite. Puisque $u_n \to b$, la relation de récurrence donne $b = \sqrt{2+b}$. En élevant au carré, on obtient $b^2 = 2+b$, soit encore $b^2 - b - 2 = 0$. Les solutions sont $b = -1$ ou $b = 2$. Or, $u_n > 1$ pour tout $n > 0$, donc $b = 2$.

- Remarque : on peut montrer que $u_n = 2\cos(\vartheta_n)$ pour tout $n \in \mathbb{N}$, ayant défini $\vartheta_n = \frac{\pi}{2^{n+1}}$. En effet, pour $n = 0$, on a bien $2\cos\left(\frac{\pi}{2}\right) = 0 = u_0$. Montrons alors que $u_{n+1} = \sqrt{2+u_n}$:

$$u_{n+1}^2 = (2\cos(\vartheta_{n+1}))^2 = 4\cos^2\left(\frac{\vartheta_n}{2}\right) = 4\frac{1+\cos(\vartheta_n)}{2} = 2 + 2\cos(\vartheta_n) = 2 + u_n$$

Exercice 4.27 (Suites type ③ : récurrence à deux pas $u_{n+1} = f(u_n, u_{n-1})$ (Fibonacci))

Un homme met un couple de lapins dans un lieu isolé. Combien de couples obtient-on en deux ans si chaque couple engendre tous les mois un nouveau couple à compter de la fin du second mois de son existence ? Par exemple, après 4 mois on aura 3 couples. On peut stocker le nombre de couples par mois dans une suite $(u_n)_{n\in\mathbb{N}}$ définie par récurrence :

$$\begin{cases} u_0 = 0, \\ u_1 = 1, \\ u_{n+2} = u_{n+1} + u_n. \end{cases}$$

Correction

Méthode avec stockage de toutes les valeurs :

```
# Avec boucle while
n = 1
u = [0,1]
while n<6:
    n += 1
    u.append(u[-1]+u[-2])
for i,ui in enumerate(u):
    print(f"u_{i}={ui}")
```

```
# Avec boucle for
u = [0,1]
for n in range(2,7):
    u.append(u[-1]+u[-2])
    #u.append(u[n-1]+u[n-2])
for i,ui in enumerate(u):
    print(f"u_{i}={ui}")
```

```
u_0=0
u_1=1
u_2=1
u_3=2
u_4=3
u_5=5
u_6=8
```

Méthode avec stockage des seules valeurs nécessaires :

```python
# Avec boucle while
n = 1
u = [0,1]
print(f"u_{0}={u[0]}")
print(f"u_{1}={u[1]}")
while n<6:
    n += 1
    u=[u[-1],u[-1]+u[-2]]
    print(f"u_{n}={u[-1]}")
```

```python
# Avec boucle for
u = [0,1]
print(f"u_{0}={u[0]}")
print(f"u_{1}={u[1]}")
for n in range(2,7):
    u = [u[-1],u[-1]+u[-2]]
    print(f"u_{n}={u[-1]}")
```

```
u_0=0
u_1=1
u_2=1
u_3=2
u_4=3
u_5=5
u_6=8
```

Notons qu'ici nous n'avons pas $u_n = f(n)$ mais $u_n = f(u_{n-1}, u_{n-2})$. Lorsqu'on aura vu le chapitre 6 sur les fonctions, on pourra alors écrire :

```python
f = lambda a,b : a+b
u = [0,1]
for n in range(2,7):
    u.append(f(u[-1],u[-2]))
for i,ui in enumerate(u):
    print(f"u_{i}={ui}")
```

```
u_0=0 u_1=1 u_2=1 u_3=2 u_4=3 u_5=5 u_6=8
```

Exercice 4.28 (Suites type ③ : récurrence à deux pas $u_{n+1} = f(u_n, u_{n-1})$ (Fibonacci – bis))

Soit $(u_n)_{n\in\mathbb{N}}$ la suite définie par récurrence :

$$\begin{cases} u_0 = 0, \\ u_1 = 1, \\ u_{n+2} = u_{n+1} + u_n. \end{cases}$$

Soit $(v_n)_{n>1}$ la suite définie par $v_n = \frac{u_n}{u_{n-1}}$. Calculer le plus petit N pour lequel $|v_N - v_{N-1}| < 10^{-10}$. On peut montrer que $v_n \xrightarrow[n\to\infty]{} \varphi$ (le nombre d'or).

Correction

```python
u = [0, 1, 1]
v = [0, 0, u[-1]/u[-2]]
n = len(u)-1
while abs(v[-1]-v[-2]) >= 1.e-10:
    u = u[1:] + [u[-1]+u[-2]]
    v = v[1:] + [u[-1]/u[-2]]
    n += 1

print(f"N = {n}")
print(v[1]-v[0])
print(v[2]-v[1])
```

Output :

```
N = 28
1.0979950282319351e-10
-4.193956293363499e-11
```

Exercice Bonus 4.29 (Pydéfis – Fibonacci)

La suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent

$$\begin{cases} F_0 = 0, \\ F_1 = 1, \\ F_{n+2} = F_{n+1} + F_n \quad \text{pour } n = 1, 2, \ldots \end{cases}$$

Défi : trouvez le premier terme de la suite de Fibonacci dont la somme des chiffres est égale à $v = 120$.
Testez votre code : si $v = 24$ alors la solution est 987 (le dix-septième terme) puisque $9 + 8 + 7 = 24$.
Source : https://pydefis.callicode.fr/defis/FiboChiffres/txt

Exercice 4.30 (Défi Turing n°2 – Fibonacci)

Chaque nouveau terme de la suite de Fibonacci est généré en ajoutant les deux termes précédents. En commençant avec 1 et 1, les 10 premiers termes sont 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.

En prenant en compte les termes de la suite de Fibonacci dont les valeurs ne dépassent pas 4 millions, trouver la somme des termes pairs.

Correction

Chaque terme de cette suite est la somme des deux termes précédents. On peut réécrire cette suite comme suit :

$$\begin{cases} F_1 = 1, \\ F_2 = 1, \\ F_{n+2} = F_{n+1} + F_n \quad \text{pour } n = 2,3,\dots \end{cases} \rightsquigarrow \begin{cases} F_0 = 0, \\ F_1 = 1, \\ F_{n+2} = F_{n+1} + F_n \quad \text{pour } n = 1,2,\dots \end{cases}$$

ainsi on a les mêmes indices qu'en Python.

Dans cet exercice on doit trouver la somme des termes de la suite de Fibonacci dont la valeur est paire et ne dépasse pas 4 millions. Un programme brute force est suffisant. Pour savoir si un nombre est pair, il faut que ce nombre soit divisible par 2, c'est à dire que le reste de la division de ce nombre avec 2 soit égal à 0. En Python, il faut utiliser le signe % pour obtenir le reste d'une division.

Soit on garde tous les termes de la suite :

```
fib = [0,1]
s = 0
while (fib[-1]<=4.e6):
    fib.append(fib[-1]+fib[-2])
    if (fib[-1]%2)==0:
        s += fib[-1]
print(s)
```

Soit on ne garde que les termes qui nous servent à construire le terme suivant :

```
a,b = 0,1
s = 0
while (b<=4.e6):
    if (b%2)==0:
        s += b
    a,b = b,a+b
print(s)
```

Dans les deux cas le résultat est

```
4613732
```

Exercice 4.31 (Suites type ③ : récurrence à deux pas $u_{n+1} = f(u_n, u_{n-1})$ (Euclide))

L'algorithme d'Euclide pour le calcul du **PGCD** de deux entiers u, v avec $u \geq v > 0$ peut être ainsi décrit. On définit, par récurrence à deux pas, une suite $(x_n)_{n\in\mathbb{N}}$ en posant $x_0 = u$, $x_1 = v$ et, tant que $x_i > 0$, on pose x_{i+1} = reste de la division euclidienne de x_{i-1} par x_i. Le dernier terme non nul de la suite est le PGCD de u et v.

Coder cet algorithme et le valider (voir aussi l'exercice 6.65).

Correction

On construit la suite

$$\begin{cases} x_0 = u, \\ x_1 = v, \\ x_{i+1} = x_{i-1} \pmod{x_i} \end{cases}$$

tant que $x_{i+1} > 0$. Dès que $x_{i+1} = 0$ on s'arrête et le PGCD de u et v sera x_i.

Soit on garde tous les termes de la suite :

```
for u,v in [(3*5,2*3),(5*7,2*3),(2*2*3,2*2)]:
    x = [u,v]
    while x[-1]!=0:
        x.append(x[-2]%x[-1])
    print(f"PGCD({u},{v})={x[-2]}")
```

```
PGCD(15,6)=3
PGCD(35,6)=1
PGCD(12,4)=4
```

Si on ne veut pas stocker toute la suite, nous pouvons alors juste écrire :

```
for u,v in [(3*5,2*3),(5*7,2*3),(2*2*3,2*2)]:
    a,b = u,v
    while b>0:
        a,b = b, a%b
    print(f"PGCD({u},{v})={a}")
```

```
PGCD(15,6)=3
PGCD(35,6)=1
PGCD(12,4)=4
```

Exercice 4.32 (Suites type ④ : récurrence $(u,v)_{n+1} = (f_1(u_n,v_n), f_2(u_n,v_n))$)
Calculer u_4 et v_4 avec $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ deux suites définies par

$$\begin{cases} u_0 = 1, \\ v_0 = -1, \\ u_{n+1} = 3v_n + 1, \\ v_{n+1} = u_n^2. \end{cases}$$

Indice : on vérifiera par exemple que $u_3 = 13$ et $v_3 = 16$.

Correction

- Méthode avec stockage des seules valeurs à l'étape n :
  - Méthode "pythonesque" avec des affectations en parallèle : l'instruction `u,v = 3*v+1, u*u` permet de définir les nouveaux u et v en même temps

```
u,v = 1,-1
print(f"u_0={u},\t v_0={v}")
for n in range(1,5):
    u,v = 3*v+1, u*u
    print(f"u_{n}={u},\t v_{n}={v}")
```

```
u_0=1,	 v_0=-1
u_1=-2,	 v_1=1
u_2=4,	 v_2=4
u_3=13,	 v_3=16
u_4=49,	 v_4=169
```

  - Méthode avec des variables supplémentaires `uold` et `vold` pour sauvegarder les anciennes valeurs

```
u,v = 1,-1
print(f"u_0={u},\t v_0={v}")
for n in range(1,5):
    uold = u
    u = 3*v+1
    v = uold*uold
    print(f"u_{n}={u},\t v_{n}={v}")
```

```
u_0=1,	 v_0=-1
u_1=-2,	 v_1=1
u_2=4,	 v_2=4
u_3=13,	 v_3=16
u_4=49,	 v_4=169
```

  - Erreur à ne pas commettre : **attention à ne pas écrire**

```
u,v=1,-1
print(f"u_0={u}, v_0={v}")
for n in range(1,5):
    u=3*v+1
    v=u*u
    print(f"u_{n}={u}, v_{n}={v}")
```

```
u_0=1, v_0=-1
u_1=-2, v_1=4
u_2=13, v_2=169
u_3=508, v_3=258064
u_4=774193, v_4=599374801249
```

qui correspond à la suite

$$\begin{cases} u_0 = 1, \\ v_0 = -1, \\ u_{n+1} = 3v_n + 1, \\ v_{n+1} = u_{\mathbf{n+1}}^2. \end{cases}$$

- Méthode avec stockage de toutes les valeurs :
  - Avec des indices explicites en fonction de l'étape n

```
u = [1]
v = [-1]
print(f"u_0={u[-1]},\t v_0={v[-1]}")
for n in range(1,5):
    u.append( 3*v[n-1]+1 )
    v.append( u[n-1]**2 )
    print(f"u_{n}={u[-1]},\t v_{n}={v[-1]}")
```

```
u_0=1,	 v_0=-1
u_1=-2,	 v_1=1
u_2=4,	 v_2=4
u_3=13,	 v_3=16
u_4=49,	 v_4=169
```

  - Avec des indices relatifs au dernier élément calculé

```
u = [1]
v = [-1]
print(f"u_0={u[-1]},\t v_0={v[-1]}")
for n in range(1,5):
    u.append( 3*v[-1]+1 )
    v.append( u[-2]**2 )
    print(f"u_{n}={u[-1]},\t v_{n}={v[-1]}")
```

```
u_0=1,	 v_0=-1
u_1=-2,	 v_1=1
u_2=4,	 v_2=4
u_3=13,	 v_3=16
u_4=49,	 v_4=169
```

- Erreur à ne pas commettre : **attention à ne pas écrire**

```python
u = [1]
v = [-1]
print(f"u_0={u[-1]},\t v_0={v[-1]}")
for n in range(1,5):
    u.append( 3*v[-1]+1 )
    v.append( u[-1]**2 )
    print(f"u_{n}={u[-1]},\t v_{n}={v[-1]}")
```

```
u_0=1,	 v_0=-1
u_1=-2,	 v_1=4
u_2=13,	 v_2=169
u_3=508,	 v_3=258064
u_4=774193,	 v_4=599374801249
```

qui correspond à la suite

$$\begin{cases} u_0 = 1, \\ v_0 = -1, \\ u_{n+1} = 3v_n + 1, \\ v_{n+1} = u_{\mathbf{n+1}}^2. \end{cases}$$

Exercice 4.33 (Suites type ⑤ : récurrence $(u,v)_{n+1} = (f_1(n,u_n,v_n), f_2(n,u_n,v_n))$)

Calculer u_{100} et v_{100} où $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ sont deux suites définies par

$$\begin{cases} u_0 = v_0 = 1, \\ u_{n+1} = u_n - \frac{v_n}{n+1}, \\ v_{n+1} = (n+1)u_n + v_n, \qquad n = 0, \dots \end{cases}$$

Hint : on vérifiera par exemple qu'on a $u_5 \approx 0.1166$ et $v_5 \approx -14.75$.

Bien noter la différence avec l'exercice 4.32 car, cette fois-ci, n intervient explicitement dans la définition de u et v.

Correction

Calculons les premiers termes de manière manuelle pour confirmer l'exactitude de notre boucle : $(u,v)_{n=0} = (1,1)$ puis $(u,v)_1 = \left(u_0 - \frac{v_0}{0+1}, (0+1)u_0 + v_0\right) = (0,2)$ et $(u,v)_2 = \left(u_1 - \frac{v_1}{1+1}, (1+1)u_1 + v_1\right) = (-1,2)$

Calcule explicite du 100-ème terme :

```python
u,v = 1,1 # n=0, u_0, v_0
for n in range(0,100):
    # NB ici !=range(1,101)
    # car on a "n" dans l'affectation
    (u,v) = (u-v/(n+1),(n+1)*u+v)
print("u_100 =",u,"\nv_100 =",v)
```

```
u_100 = -82506365759177.17
v_100 = -9476942524736076.0
```

Exercice 4.34 (La suite de Stern-Brocot)

La suite diatomique de Stern-Brocot est définie ainsi :

$$\begin{cases} s_0 = 0, \\ s_1 = 1, \\ s_{2n} = s_n, \\ s_{2n+1} = s_n + s_{n+1} \end{cases}$$

Les premiers éléments de la suite sont 0, 1, 1, 2, 1, 3, 2, 3, 1, 4, 3. Que vaut s_{100} ?

Correction

Source : https://www.cristal.univ-lille.fr/~jdelahay/pls/227.pdf

```python
N = 100
s = [0, 1]
for n in range(2, N + 1):
    if n % 2 == 0:
        s.append(s[n // 2])
    else:
        s.append(s[n // 2] + s[n // 2 + 1])

print(s[-1])
```

```
7
```

```python
N = 100
```

```
s = [ 0 for i in range(N+1) ]
s[1] = 1
for n in range(2, N + 1):
    if n % 2 == 0:
        s[n] = s[n // 2]
    else:
        s[n] = s[n // 2] + s[n // 2 + 1]

print(s[N])
```

```
7
```

Exercice 4.35 (La suite G de Hofstadter)

La G-Suite est définie ainsi :

$$\begin{cases} G_0 = 0, \\ G_n = n - G_{G_{n-1}} \quad \text{pour } n > 0. \end{cases}$$

On remarque que le calcul de G_n fait intervenir le terme G_{n-1} en **tant qu'indice**. On peut montrer que $G_{n-1} < n$ ainsi le terme $G_{G_{n-1}}$ est disponible à l'itération n.

Les premiers éléments de la suite sont 0, 1, 1, 2, 3, 3, 4, 4, 5, 6. Calculer G_{76}.

Correction

https://oeis.org/A005206

$$\begin{aligned}
G_0 &= 0 \quad \text{(défini)} \\
G_1 &= 1 - G_{G_0} = 1 - G_0 = 1 - 0 = 1 \\
G_2 &= 2 - G_{G_1} = 2 - G_1 = 2 - 1 = 1 \\
G_3 &= 3 - G_{G_2} = 3 - G_1 = 3 - 1 = 2 \\
G_4 &= 4 - G_{G_3} = 4 - G_2 = 4 - 1 = 3 \\
G_5 &= 5 - G_{G_4} = 5 - G_3 = 5 - 2 = 3 \\
G_6 &= 6 - G_{G_5} = 6 - G_3 = 6 - 2 = 4 \\
G_7 &= 7 - G_{G_6} = 7 - G_4 = 7 - 3 = 4 \\
G_8 &= 8 - G_{G_7} = 8 - G_4 = 8 - 3 = 5 \\
G_9 &= 9 - G_{G_8} = 9 - G_5 = 9 - 3 = 6
\end{aligned}$$

```
N = 76
G = [0]
for n in range(1, N + 1):
    G.append(n-G[G[n-1]])
print(f"{n = }, {G[-1] = }")
```

```
n = 76, G[-1] = 47
```

On peut montrer que $G_n = \lfloor \frac{n+1}{\varphi} \rfloor$, φ étant le nombre d'or. Vérifions-le avec python :

```
N = 76
phi = (1+5**(0.5))/2
g = int( (N+1)/phi )
print(f"{N = }, {g = }")
```

```
N = 76, g = 47
```

Exercice 4.36 (La suite H de Hofstadter)

La H-Suite est définie ainsi :

$$\begin{cases} H_0 = 0, \\ H_n = n - H_{H_{H_{n-1}}} \quad \text{pour } n > 0. \end{cases}$$

Les premiers éléments de la suite sont 0, 1, 1, 2, 3, 4, 4, 5, 5, 6. Calculer G_{73}.

Correction

https://oeis.org/A005374

$$
\begin{aligned}
&H_0 = 0 \quad \text{(défini)}\\
&H_1 = 1 - H_{H_{H_0}} = 1 - H_{H_0} = 1 - H_0 = 1 - 0 = 1\\
&H_2 = 2 - H_{H_{H_1}} = 2 - H_{H_1} = 2 - H_1 = 2 - 1 = 1\\
&H_3 = 3 - H_{H_{H_2}} = 3 - H_{H_1} = 3 - H_1 = 3 - 1 = 2\\
&H_4 = 4 - H_{H_{H_3}} = 4 - H_{H_2} = 4 - H_1 = 4 - 1 = 3\\
&H_5 = 5 - H_{H_{H_4}} = 5 - H_{H_3} = 5 - H_2 = 5 - 1 = 4\\
&H_6 = 6 - H_{H_{H_5}} = 6 - H_{H_4} = 6 - H_3 = 6 - 2 = 4\\
&H_7 = 7 - H_{H_{H_6}} = 7 - H_{H_4} = 7 - H_3 = 7 - 2 = 5\\
&H_8 = 8 - H_{H_{H_7}} = 8 - H_{H_5} = 8 - H_4 = 8 - 3 = 5\\
&H_9 = 9 - H_{H_{H_8}} = 9 - H_{H_5} = 9 - H_4 = 9 - 3 = 6
\end{aligned}
$$

```python
N = 73
H = [0]
for n in range(1, N + 1):
    H.append(n-H[H[H[n-1]]])
print(f"{n = }, {H[-1] = }")

n = 73, H[-1] = 50
```

Exercice 4.37 (La suite Female-Male de Hofstadter)

La FM-Suite est définie ainsi :

$$
\begin{cases}
M_0 = 0,\\
F_0 = 1,\\
M_n = n - F_{M_{n-1}},\\
F_n = n - M_{F_{n-1}}.
\end{cases}
$$

Les premiers éléments de la suite M sont 0, 0, 1, 2, 2, 3, 4, 4, 5, 6, ceux de la suite F sont 1, 1, 2, 2, 3, 3, 4, 5, 5, 6. Calculer $(M,F)_{73}$.

Correction

https://oeis.org/A005378 et https://oeis.org/A005379

$$
\begin{aligned}
&(M_0, F_0) = (0, 1) \quad \text{(défini)}\\
&(M_1, F_1) = (1 - F_{M_0},\ 1 - M_{F_0}) = (1 - F_0,\ 1 - M_1) = (0,\ 1)\\
&(M_2, F_2) = (2 - F_{M_1},\ 2 - M_{F_1}) = (2 - F_0,\ 2 - M_1) = (1,\ 2)\\
&(M_3, F_3) = (3 - F_{M_2},\ 3 - M_{F_2}) = (3 - F_1,\ 3 - M_2) = (2,\ 2)\\
&(M_4, F_4) = (4 - F_{M_3},\ 4 - M_{F_3}) = (4 - F_2,\ 4 - M_2) = (2,\ 3)\\
&(M_5, F_5) = (5 - F_{M_4},\ 5 - M_{F_4}) = (5 - F_2,\ 5 - M_3) = (3,\ 3)\\
&(M_6, F_6) = (6 - F_{M_5},\ 6 - M_{F_5}) = (6 - F_3,\ 6 - M_3) = (4,\ 4)\\
&(M_7, F_7) = (7 - F_{M_6},\ 7 - M_{F_6}) = (7 - F_4,\ 7 - M_4) = (4,\ 5)\\
&(M_8, F_8) = (8 - F_{M_7},\ 8 - M_{F_7}) = (8 - F_4,\ 8 - M_5) = (5,\ 5)\\
&(M_9, F_9) = (9 - F_{M_8},\ 9 - M_{F_8}) = (9 - F_5,\ 9 - M_5) = (6,\ 6)
\end{aligned}
$$

```python
N = 73
(M,F) = ([0],[1])
for n in range(1, N + 1):
    M.append(n-F[M[n-1]])
    F.append(n-M[F[n-1]])
print(f"{n = }, {M[-1] = }, {F[-1] = }")

n = 73, M[-1] = 45, F[-1] = 45
```

Exercice 4.38 (La suite Hofstadter Figure-Figure (R et S))

Les suites R et S sont définies par

$$
\begin{cases}
R_1 = 1,\\
S_1 = 2,\\
R_n = R_{n-1} + S_{n-1} \quad \text{pour } n > 1,\\
S_n = \text{ plus petit entier} > \text{à } S_{n-1} \text{ et absents de R.}
\end{cases}
$$

Les suites R et S sont complémentaires : tous les entiers positifs sont dans l'un ou l'autre. Les premiers éléments de

la suite R sont 1, 3, 7, 12, 18, 26, 35, 45, 56, 69, ceux de la suite S sont 2, 4, 5, 6, 8, 9, 10, 11, 13, 14. Calculer $(R,S)_{50}$.

Correction
https://oeis.org/A005228 et https://oeis.org/A030124

$$\begin{aligned}
R_1 &= 1, & S_1 &= 2,\\
R_2 &= R_1 + S_1 = 1+2 = 3, & S_2 &= \text{plus petit entier} > S_1 \text{ et non dans } \{1,3\} = 4,\\
R_3 &= R_2 + S_2 = 3+4 = 7, & S_3 &= \text{plus petit entier} > S_2 \text{ et non dans } \{1,3,7\} = 5,\\
R_4 &= R_3 + S_3 = 7+5 = 12, & S_4 &= \text{plus petit entier} > S_3 \text{ et non dans } \{1,3,7,12\} = 6
\end{aligned}$$

```python
N = 50
R, S = [1], [2]

def next_S(S, R):
    candidate = S[-1] + 1
    while candidate in R:
        candidate += 1
    return candidate

for n in range(2, N+1):
    R.append(R[-1] + S[-1])
    S.append(next_S(S, R))

print(f"R({N}) = {R[-1]}, S({N}) = {S[-1]}")
```

```
R(50) = 1509, S(50) = 59
```

Exercice Bonus 4.39 (Pydéfis – La suite Q de Hofstadter)

La Q-Suite est définie ainsi :

$$\begin{cases} Q_1 = 1, \\ Q_2 = 1, \\ Q_n = Q_{n-Q_{n-1}} + Q_{n-Q_{n-2}} \quad \text{pour } n > 2. \end{cases}$$

Que vaut $\sum_{i=2313}^{i=2375} Q_i$?

Source : https://pydefis.callicode.fr/defis/QSuite/txt

Exercice Bonus 4.40 (La suite de Hofstadter-Conway)

La suite de Hofstadter-Conway est définie ainsi :

$$\begin{cases} a_1 = 1, \\ a_2 = 1, \\ a_n = a_{a_{n-1}} + a_{n-a_{n-1}} \quad \text{pour } n > 2. \end{cases}$$

Que vaut a_N pour $N = 10^4$?

Source : https://codex.forge.apps.education.fr/exercices/HC10000/#grandes-valeurs-de-n

Exercice 4.41 (S-T-R-E-T-C-H I-T O-U-T)

Étant donné une liste de nombres x et un entier n, écrire une fonction qui “étire” la liste de nombres en insérant n nombres entre chaque paire de nombres dans x, en utilisant une interpolation linéaire. Autrement-dit, entre deux valeurs consécutives x_{i-1} et x_i, on insère n valeurs régulièrement espacées.

Par exemple, si $x = [1,5,2,4,3]$ et $n = 1$, la liste obtenue est $y = [1,3,5,3.5,2,3,4,3.5,3]$:

$$\begin{array}{lccccccccccc}
x = [& 1, & & 5, & & 2, & & 4, & & 3 &] \\
 & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\
y = [& 1, & 3, & 5, & 3.5, & 2, & 3, & 4, & 3.5, & 3 &]
\end{array}$$

Exemple avec la même liste et $n = 2$ (on insère deux valeurs entre chaque paire consécutive) :

$$\begin{array}{lcccccccccccccc} x = [& 1, & & & 5, & & & 2, & & & 4, & & & 3 &] \\ & \downarrow & & & \downarrow & & & \downarrow & & & \downarrow & & & \downarrow & \\ y = [& 1, & \frac{7}{3}, & \frac{11}{3}, & 5, & \frac{13}{3}, & \frac{11}{3}, & 2, & \frac{8}{3}, & \frac{10}{3}, & 4, & \frac{11}{3}, & \frac{10}{3}, & 3 &] \end{array}$$

Correction

Le pas d'interpolation est donné par $h = \frac{x_i - x_{i-1}}{n+1}$. Les nouvelles valeurs à ajouter entre x_{i-1} et x_i sont alors

$$x_{i-1} + h, \quad x_{i-1} + 2h, \quad \ldots, \quad x_{i-1} + nh.$$

Par exemple, pour $x = [1, 5, 2, 4, 3]$ et $n = 1$, on obtient $y = [1, 3, 5, 3.5, 2, 3, 4, 3.5, 3]$.

- Entre 1 et 5, on a $h = (5-1)/2 = 2$, ce qui donne $1, 3, 5$.
- Entre 5 et 2, on a $h = (2-5)/2 = -1.5$, ce qui donne $5, 3.5, 2$.
- Entre 2 et 4, on a $h = (4-2)/2 = 1$, ce qui donne $2, 3, 4$.
- Entre 4 et 3, on a $h = (3-4)/2 = -0.5$, ce qui donne $4, 3.5, 3$.

Avec la même liste x et $n = 2$ on obtient $y = [1, \frac{7}{3}, \frac{11}{3}, 5, \frac{13}{3}, \frac{11}{3}, 2, \frac{8}{3}, \frac{10}{3}, 4, \frac{11}{3}, \frac{10}{3}, 3]$.

- Entre 1 et 5, on a $h = (5-1)/3 = \frac{4}{3}$, ce qui donne $1, \frac{7}{3}, \frac{11}{3}, 5$.
- Entre 5 et 2, on a $h = (2-5)/3 = -1$, ce qui donne $5, 4, 3, 2$.
- Entre 2 et 4, on a $h = (4-2)/3 = \frac{2}{3}$, ce qui donne $2, \frac{8}{3}, \frac{10}{3}, 4$.
- Entre 4 et 3, on a $h = (3-4)/3 = -\frac{1}{3}$, ce qui donne $4, \frac{11}{3}, \frac{10}{3}, 3$.

```python
x = [1, 5, 2, 4, 3]
n = 1

y = [x[0]]
for i in range(1, len(x)):
    h = (x[i] - x[i-1]) / (n + 1)
    for j in range(1, n + 2):
        y.append(x[i-1] + j * h)

print(y)
```

```
[1, 3.0, 5.0, 3.5, 2.0, 3.0, 4.0, 3.5, 3.0]
```

Exercice 4.42 (Nombre mystère)

Trouvez le nombre mystère $n \in \mathbb{N}$ qui répond aux conditions suivantes :

- C'est un entier positif pair composé de 3 chiffres et strictement inférieur à 300.
- La somme de ses chiffres est égale à 7.
- Au moins deux de ses chiffres sont identiques.

Correction

Première méthode : on considère tous les entiers pairs compris entre 100 et 299 et on vérifie s'ils satisfont les deux dernières propriétés.

```python
for nombre in range(100, 300, 2):
    chiffres = [] # liste de int
    for ch in str(nombre):
        chiffres.append(int(ch))
    if sum(chiffres) == 7 and len(set(chiffres)) < 3 :
        print(nombre)
```

```
232
```

- **Boucle** `for` : cette boucle itère sur les nombres de 100 à 299 (inclus), en sautant de 2 en 2, ce qui garantit que les nombres générés seront pairs.
- **Liste des chiffres** : on convertit chaque chiffre du nombre en une liste d'entiers. Au prochain chapitre on verra une façon plus compact de construire la même liste :

  ```python
  chiffres = [int(ch) for ch in str(nombre)]
  ```

- **Conditions de vérification** :
  `sum(chiffres) == 7` vérifie si la somme des chiffres est égale à 7.
  `len(set(chiffres)) < 3` vérifie si le nombre a moins de 3 chiffres différents (et donc a au moins 2 chiffres identiques). En effet, `set(L)` crée un ensemble à partir de la liste L. Un ensemble ne peut pas contenir de doublons, donc s'il y a des doublons dans la liste, la longueur de l'ensemble sera inférieure à la longueur de la liste, ici 3.

Deuxième méthode : on construit le nombre sous la forme $n = c \times 10^2 + d \times 10 + u$:
- L'entier est composé de trois chiffres ($c > 0$) et est strictement inférieur à 300 ($c < 3$) : $c \in \{1,2\}$.
- Il est pair : $u \in \{0,2,4,6,8\}$, ce qui peut s'écrire comme `range(0, 9, 2)`.
- La somme de ses chiffres est égale à 7 : $c + d + u = 7$
- Au moins deux de ses chiffres sont identiques : `len(set([c,d,u]))<=2`

```
for c in [1,2]:
    for u in range(0, 9, 2):
        d = 7-u-c
        if len(set([c,d,u]))<=2:
            print(c*100+d*10+u)
```

```
232
```

Exercice 4.43 (4n)

Considérons un nombre naturel m se terminant par le chiffre 4. Pour illustrer cette terminaison par 4, écrivons ce nombre sous la forme $\underline{s}4$, où $\underline{s}$ représente la séquence des chiffres constituant m dans son écriture décimale, à l'exception du dernier chiffre, qui est 4. Par exemple, si $m = 125674$, alors $\underline{s} = 12567$.

Parmi tous les nombres se terminant par 4, il en existe un pour lequel déplacer le chiffre 4 tout à gauche produit un nouveau nombre n égal à quatre fois la valeur initiale de m. En d'autres termes, si l'on représente par $4\underline{s}$ la séquence des chiffres constituant n dans son écriture décimal, alors $n = 4 \times m$. Qui est m ?

Correction

Le nombre m se termine par le chiffre 4, donc on peut le construire comme $10s + 4$ avec $s \in \mathbb{N}$. Le nombre n obtenu en déplaçant le dernier 4 tout à gauche peut être construit comme `int('4'+str(m)[:-1])`. On cherche le nombre m tel que $4m = n$.

```
m = 14
while 4*m != int(str(4)+str(m)[:-1]):
    m += 10

print(f'{m = } et {4*m = }')
```

```
m = 102564 et 4*m = 410256
```

Exercice Bonus 4.44 (Défi Turing n°70 – Permutation circulaire)

Prenons le nombre $n = 102564$. En faisant passer le dernier chiffre complètement à gauche (*i.e.* on enlève le chiffre des unités et on le place toute à gauche), on obtient un nouveau nombre : $m = 410256$. Ce nouveau nombre a une particularité : il est un multiple du nombre de départ (mais différent du nombre de départ). En effet, $m = 410256 = 102564 \times 4 = 4n$.

Additionner tous les nombres de 6 chiffres ayant cette propriété.

Remarque : dans l'exercice 4.43 on cherchait que les multiples de 4, ici n'importe quel multiple.

Correction

```
L = []
for n in range(10**5,10**6):
    m = int(str(n)[-1]+str(n)[:-1]) # Walrus:  m = int( (s := str(n))[-1] + s[:-1] )
    if m!=n and m%n==0 :
        L.append(n)
        print(f"{n = } car {n} x {m//n} = {m}")
# print("Les nombres ayant cette propriété sont", *L)
print("Leur somme vaut", sum(L))
```

```
n = 102564 car 102564 x 4 = 410256
n = 128205 car 128205 x 4 = 512820
n = 142857 car 142857 x 5 = 714285
n = 153846 car 153846 x 4 = 615384
n = 179487 car 179487 x 4 = 717948
n = 205128 car 205128 x 4 = 820512
n = 230769 car 230769 x 4 = 923076
Leur somme vaut 1142856
```

Exercice Bonus 4.45 (Gator And Ducks)

Des canards nagent en cercle au bord d'un étang. Cet étang abrite également un alligator qui raffole de canards. En partant d'une position donnée (le canard n°1), l'alligator compte dans le sens des aiguilles d'une montre autour du cercle et mange le k^e canard. Après chaque disparition, le cercle se referme et le comptage reprend à partir du canard suivant. On continue ainsi jusqu'à ce que tous les canards aient été mangés.

Écrire un script qui, étant donné :

- n : le nombre de canards,
- k : le pas de comptage,

renvoie les deux listes suivantes :

- l'ordre dans lequel les canards sont mangés ;
- pour chaque canard, son rang de consommation.

Source : https://py.checkio.org/en/mission/gator-and-ducks/

◊

Par exemple, lorsque le nombre de canards est $n = 8$ et que le pas est $k = 4$, les canards sont mangés dans l'ordre suivant :

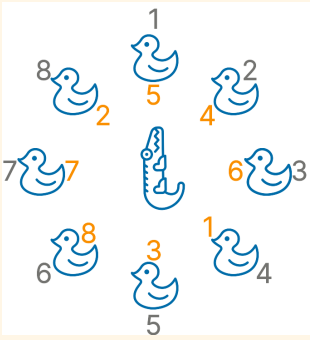
$$4, 8, 5, 2, 1, 3, 7, 6.$$

Cela signifie que le premier canard mangé est le n°4, le deuxième le n°8, le troisième le n°5, etc.

On peut également décrire le menu de l'alligator autrement : pour chaque canard, on indique son *rang de consommation*. Dans cet exemple :

$$[5, 4, 6, 1, 3, 8, 7, 2]$$

signifie que le canard n°1 est mangé en 5e, le canard n°2 est mangé en 4e, le canard n°3 est mangé en 6e, le canard n°4 est mangé en 1er, etc.



- Entrée : $n = 8$, $k = 4$;
- Sortie :

```
ordre_consommation = [4,8,5,2,1,3,7,6]
rang_par_canard    = [5,4,6,1,3,8,7,2]
```

◊

Variante (Défi Turing n°60 : Suicide collectif) : les 2013 membres d'une secte ont décidé de se suicider. Pour effectuer le rituel funèbre, ils se mettent en cercle, puis se numérotent dans l'ordre de de 1 à 2013. On commence à compter, à partir du numéro 1. Toutes les 7 positions, la personne désignée devra mourir. Ainsi, la première à mourir aura le n°7, la deuxième le 14, la troisième le 21, etc. Vous faites partie de cette secte, mais vous n'avez aucune envie de mourir ! Il s'agit donc de trouver la position sur le cercle qui vous permettra d'être désigné en dernier, et donc d'échapper à la mort. Quelle est la position qui vous sauvera ?

Correction

```python
n = 8; k = 4

ducks = list(range(1,n+1))
ordre = []
rang = [0]*n

i, step = 0, 1

while ducks:
    i = (i + k - 1) % len(ducks)
    d = ducks.pop(i)
    ordre.append(d)
    rang[d-1] = step
    step += 1
```

```
print("Ordre de Consommation =", ordre)
print("Rang par Canard =", rang)
```

```
Ordre de Consommation = [4, 8, 5, 2, 1, 3, 7, 6]
Rang par Canard = [5, 4, 6, 1, 3, 8, 7, 2]
```

Dans la variante du défi Turing, on cherche seulement la dernière position restante.

```
n = 2013; k = 7

L = list(range(1,n+1))
i = 0

while len(L) > 1:
    i = (i + k - 1) % len(L)
    del L[i]

print(L[0])
```

```
1868
```

Exercice Bonus 4.46 (Défis Turing n°72)
Parmi tous les entiers inférieurs à 1 milliard, combien sont des carrés se terminant par exactement 3 chiffres identiques (mais pas 4 ou plus) ? Par exemple, $213444 = 462^2$.

Correction
On peut calculer le carrée des nombres de 10 à 10^5 et vérifier qu'il a au moins 3 chiffres et au plus 9 chiffres (*i.e.* $10^3 \leq n < 10^9$), que les derniers trois chiffres sont identiques mais pas celui qui les précède.

```
L = {}
for i in range(4,10**5):
    n = i*i
    sn = str(n)
    if n<=10**9 and sn[-1]==sn[-2]==sn[-3] and sn[-4]!=sn[-3]:
        L[i] = n
#print(L)
print(len(L))
```

```
127
```

La condition `sn[-1]==sn[-2]==sn[-3]` équivaut à `len(set(sn[-3:]))==1`.

Exercice Bonus 4.47 (Dictionnaire)
Soit `L` une liste de listes. Créer un dictionnaire qui contient comme clés la longueur des listes et comme valeur la liste des listes de telle longueur. Autrement dit, on regroupe chaque liste de la liste `L` selon leur longueur.

Correction

```
L = [ [1], [4,5], [-1,8,9], [11,12,13] , [0] , [1,2] , [3,17] ]

dp = {}
for l in L:
    k = len(l)
    if k not in dp:
        dp[k] = []
    dp[k].append(l)

print(dp)
```

```
{1: [[1], [0]], 2: [[4, 5], [1, 2], [3, 17]], 3: [[-1, 8, 9], [11, 12, 13]]}
```

Exercice Bonus 4.48 (Dictionnaires : construction d'un histogramme)
Supposons que nous voulions créer un histogramme de la fréquence d'utilisation de chaque lettre de l'alphabet dans un texte donné. Il est possible de réaliser cette tâche avec un algorithme extrêmement simple basé sur un dictionnaire.

Tout d'abord, nous créons un dictionnaire vide nommé `lettres`. Ensuite, nous remplissons le dictionnaire en utilisant les caractères de l'alphabet comme clés. Les valeurs stockées pour chacune de ces clés sont les fréquences des caractères correspondants dans le texte. Pour calculer ces fréquences, nous parcourons la chaîne de caractères

texte. Pour chaque caractère, nous utilisons la méthode `get()` pour interroger le dictionnaire en utilisant le caractère comme clé. Ainsi, nous pouvons lire la fréquence déjà stockée pour ce caractère. Si cette valeur n'existe pas encore, la méthode `get()` renvoie une valeur nulle. Dans tous les cas, nous incrémentons la valeur trouvée et la stockons dans le dictionnaire à l'emplacement correspondant à la clé (c'est-à-dire au caractère en cours de traitement).

Enfin, si nous souhaitons afficher l'histogramme dans l'ordre alphabétique, nous pouvons convertir le dictionnaire en une liste de tuples. Nous pouvons alors utiliser la méthode `sort()`, qui ne s'applique qu'aux listes, pour trier cette liste.

Source : https://python.developpez.com/cours/apprendre-python3/?page=page_12

Correction

```
texte = "les saucisses et saucissons secs sont dans le saloir"
lettres = {}
for c in texte:
    lettres[c] = lettres.get(c, 0) + 1
print(lettres)
```

```
{'l': 3, 'e': 5, 's': 14, ' ': 8, 'a': 4, 'u': 2, 'c': 3, 'i': 3, 't': 2, 'o': 3, 'n': 3, 'd': 1, 'r':
 ↪  1}
```

```
lettres_triees = list(lettres.items())
lettres_triees.sort()
print(lettres_triees)
```

```
[(' ', 8), ('a', 4), ('c', 3), ('d', 1), ('e', 5), ('i', 3), ('l', 3), ('n', 3), ('o', 3), ('r', 1),
 ↪  ('s', 14), ('t', 2), ('u', 2)]
```

Exercice Bonus 4.49 (Dictionnaires : The Most Frequent)

Vous avez une séquence de chaînes et vous souhaitez déterminer la chaîne la plus fréquente dans la séquence (il n'y en a qu'une).

Entrée : une liste non vide de chaînes de caractères.

Sortie : une chaîne de caractères.

Source : https://py.checkio.org/en/mission/the-most-frequent/

Correction

```
def most_frequent(data: list[str]) -> str:
    dico = {}
    for d in data:
        dico[d] = dico.get(d,0)+1
    return max(dico,key=dico.get)

# TESTS
print(most_frequent(["a", "b", "c", "a", "b", "a"]))
print(most_frequent(["a", "a", "bi", "bi", "bi", "c"]))
print(most_frequent(["a"]))
print(most_frequent(["a","a","z"]))
```

```
a
bi
a
a
```

Exercice Bonus 4.50 (Pydéfis – Vous parlez Fourchelangue ?)

En ce moment Harry fait des rêves inquiétants et il arrive même qu'il parle Fourchelangue pendant son sommeil.

Le Fourchelangue n'est finalement pas très difficile à comprendre : chaque « syllabe » Fourchelangue correspond à un caractère. La correspondance est donnée dans la table suivante (on remarque que I et J d'une part, et U et V d'autre part se prononcent de la même manière en Fourchelangue) :

HFH	FFH	SHS	SHH	SSH	FHF	FSS	HFF	HHH	SFS	FFS	FHS
A	B	C	D	E	F	G	H	IJ	K	L	M
SSF	FHH	HHF	SFF	FSF	FSH	HHS	FFF	SSS	HFS	SHF	SFH
N	O	P	Q	R	S	T	UV	W	X	Y	Z

La syllabe HS permet de séparer les mots.

À titre d'exemple, `BONJOUR HARRY` se dit, en Fourchelangue, `FFHFHHSSFHHHFHHFFFFSFHSHFFHFHFSFFSFSHF`.

Ron a réussi à noter ce que Harry a dit la nuit dernière. Il demande votre aide pour comprendre le contenu du message car il craint que certaines personnes ne soient en danger.

Le texte à traduire est disponible ici : https://pydefis.callicode.fr/defis/Fourchelangue/txt

Exercice Bonus 4.51 (Pydéfis - Code Konami)

Dans le monde dédié à Gradius, vous affronterez de multiples ennemis aux commandes du Vic Viper. Un message vous a été remis au début de votre mission, qui devrait vous révéler les épreuves qui vous attendent. Ce message semble directement inspiré du Code Konami.

Des membres de votre équipe ont pu mettre la main sur le moyen de traduire ces messages. Le code est composé de 6 symboles : ←, →, ↓, ↑, A et B. À chaque couple de symboles (il y a 36 possibilités) est associé un caractère alphabétique ou de ponctuation. Par exemple : BA correspond à e, ←→ correspond à m, ↓↓ correspond à o, ↑↑ correspond à p. Ainsi, la séquence ↑↑↓↓←→←→BA en Code Konami correspond à "pomme".

Le code complet ainsi que l'intégralité du message à déchiffrer sont disponibles ici : https://pydefis.callicode.fr/defis/C22_KonamiCode/input

Ce message vous révèle, entre autres, quelle sera la sixième épreuve à affronter. Indiquez-la pour valider le défi (entrez moins de 50 caractères).

Source : https://pydefis.callicode.fr/defis/C22_KonamiCode/txt

Exercice Bonus 4.52 (Pydéfis - Difficile de comprendre un lapin crétin)

Vous ne vouliez pas y aller, mais vous venez de vous retrouver dans le monde des lapins crétins. Ça se termine généralement en explosion et catastrophe, et vous ne comptez pas perdre votre avatar si bêtement.

Tic tac... Ils jouent encore avec une bombe.

Vous devez la désamorcer avant qu'il ne soit trop tard, mais elle est remplie de leviers, manettes... et vous n'avez pas d'autre instruction qu'un lapin qui semble vouloir votre bien et vous informe de ce qu'il faut faire... à sa façon. BWAXA BWAWA ?

Vous savez qu'ils ne disent pas n'importe quoi. Les lettres utiles sont simplement insérées entre des syllabes sans signification, de type : `BWA.A` où `.` peut être une lettre arbitraire.

Pour dire HELLO, un lapin pourrait «simplement» vous dire :

```
BWAYABWANAHBWAIAEBWAPABWAMABWAZALBWAPABWALALBWAGABWAQABWAEAOBWAEABWAYA
          ^     ^               ^          ^               ^
```

Si on découpe les syllabes, la structure est très visible :

```
BWAYA BWANA H BWAIA E BWAPA BWAMA BWAZA L BWAPA BWALA L BWAGA BWAQA BWAEA O BWAEA BWAYA
            ^       ^                   ^             ^                   ^
```

La transcription de votre ami le lapin est disponible ici : https://pydefis.callicode.fr/defis/C22_BwaCode01/input

Sur quoi devez-vous appuyer pour désamorcer la bombe ?

Source : https://pydefis.callicode.fr/defis/C22_BwaCode01/txt

Exercice Bonus 4.53 (Pydéfis – Le retourneur de temps)

Le professeur Dumbledore a confié à Hermione son retourneur de temps pour l'aider à suivre tous les cours qu'elle a choisis pour sa troisième année.

Le retourneur de temps est un objet très complexe. Il permet de remonter dans le temps d'un certain nombre de minutes. Pour voyager dans le passé, on fait faire des tours au retourneur de temps (il ressemble à un sablier), et à chaque tour, une quantité de minutes à remonter est calculée. Lorsqu'on arrête de l'actionner, le voyage commence.

- En faisant un seul tour, le retourneur nous enverra 2 minutes dans le passé.
- En faisant deux tours, il nous enverra 4 minutes dans le passé
- Avec trois tours, nous ferons un voyage de 6 minutes dans le passé
- ...

On a l'impression qu'à chaque tour, le voyage nous fera reculer de 2 minutes supplémentaires. Mais le principe est en fait un peu plus complexe.

Si à un moment le nombre de minutes du voyage a la somme de ses chiffres divisible par 7 (c'est-à-dire si la somme de ses chiffres vaut 7, ou 14, ou 21...), alors au prochain tour, plutôt que de nous faire voyager de 2 minutes supplémentaires dans le passé, le retourneur modifiera la durée pour que notre voyage nous fasse remonter de 7 minutes de moins (au lieu de 2 de plus) !

Le tableau suivant récapitule la durée du voyage (en minutes) en fonction du nombre de tours effectués sur le retourneur :

nombre de tours	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
durée voyage	2	4	6	8	10	12	14	**16**	9	11	13	15	17	19	21	23	**25**	18

Comme la somme des chiffres de 16 est divisible par 7, le 9ème tour, au lieu de nous faire remonter le temps de 16 + 2 minutes, nous le fera remonter de 16-7 minutes.

De même, comme la somme des chiffres de 25 est divisible par 7, le 18ème tour, au lieu de nous faire remonter le temps de 25 + 2 minutes, nous le fera remonter de 25-7 minutes.

Le retourneur de temps est ainsi conçu pour qu'on ne puisse pas voyager trop longtemps dans le passé. Cette manière de calculer impose en effet une limite pour le voyage le plus long qu'il est possible de faire.

Combien de minutes on peut reculer dans le temps au maximum ?

Source : https://pydefis.callicode.fr/defis/RetourneurTemps/txt

Exercice Bonus 4.54 (Pydéfis – Entrée au Ministère)

Il n'est pas si facile d'entrer au ministère de la magie. Un des moyens d'accès, outre la poudre de cheminette (et les cuvettes des toilettes), est de passer par une des cabines téléphoniques Londoniennes. Une fois dans la cabine, on entre un code et après un rapide voyage, on se retrouve dans l'atrium du ministère.

Le code de la cabine est changé régulièrement, mais il a toujours la même caractéristique. Si on note les chiffres utilisés pour écrire le carré du code, on a très exactement besoin des chiffres 1, 2, 4, 6 et 7. Ainsi, récemment, le code à entrer dans la cabine était 64224. En effet, $64224^2 = 4124722176$. On a besoin des chiffres 1, 2, 4, 6 et 7 pour l'écrire. Le prochain code est le prochain nombre, plus grand que 64224, qui a la même propriété, et ainsi de suite. Notez que 64631 ne convient pas, car une fois mis au carré, il ne contient pas de 2. Or le carré doit utiliser exactement les chiffres 1, 2, 4, 6 et 7 (tous !). Quels seront les trois prochains codes à utiliser dans la cabine pour entrer au ministère de la magie ?

Source : https://pydefis.callicode.fr/defis/CodeCabine/txt

Exercice Bonus 4.55 (Pydéfis – Créatures nocturnes pas si sympathiques que cela...)

Antonio a eu l'idée saugrenue d'aller se promener dans une forêt inconnue une nuit de pleine lune. Bien entendu, il ne savait pas que celle-ci était peuplée de créatures nocturnes pour le moins antipathiques. En effet, il est amené à rencontrer des meutes de chauves-souris enragées, des skellingtons et des zombies tout droit sortis de Minecraft et Call of Duty ainsi que des fantômes baveux.

Heureusement, Antonio a toujours une faux dans son sac. Oui, c'est étrange, mais qu'est-ce qui ne l'est pas dans un jeu de survie ! Cette faux va lui permettre de faucher plus ou moins rapidement ces créatures maléfiques de la nuit et même, qui sait, de lui sauver la vie...

Sachant que :

- au démarrage du jeu, il n'y a aucune créature ;
- 10 chauves-souris apparaissent toutes les 2 secondes ;
- 5 skellingtons apparaissent toutes les 5 secondes ;
- 4 zombies apparaissent toutes les 6 secondes ;
- 3 fantômes baveux apparaissent toutes les 10 secondes.

Durant les quatre premières minutes de jeu il faut à Antonio :

- 6 secondes pour tuer 2 chauves-souris ;
- 20 secondes pour tuer 1 skellington ;
- 30 secondes pour tuer 1 zombie ;
- 40 secondes pour tuer 1 fantôme baveux.

Toutes les quatre minutes Antonio va s'améliorer, et dans le même temps, il pourra tuer 2 chauves-souris, 1 skellington, 1 zombie et 1 fantôme baveux supplémentaires.

Voici comment évoluent le nombre de chauves-souris, de skellingtons, de zombies et de fantômes baveux au fil

des secondes :

```
au départ                :  0, 0, 0, 0
au bout de   1 seconde   :  0, 0, 0, 0
au bout de   2 secondes : 10, 0, 0, 0   <= chauves-souris +10
au bout de   3 secondes : 10, 0, 0, 0
au bout de   4 secondes : 20, 0, 0, 0   <= chauves-souris +10
au bout de   5 secondes : 20, 5, 0, 0   <= skellingtons +5
au bout de   6 secondes : 28, 5, 4, 0   <= chauves-souris +10 -2
                                           zombies +4
...
au bout de 238 secondes : 1112, 224, 149, 64   <= chauves-souris +10
au bout de 239 secondes : 1112, 224, 149, 64
au bout de 240 secondes : 1120, 228, 152, 66   <= chauves-souris +10 -2
                                                   skellingtons +5 -1
                                                   zombies +4 -1
                                                   fantômes +3 -1

À partir de maintenant, Antonio tuera 4 chauves-souris en 6 secondes, 2 skellingtons
en 20 secondes, 2 zombies en 30 secondes et 2 fantômes baveux en 40 secondes.
```

Au bout de 50 minutes de jeu, combien restera-t-il de chauves-souris, de skellingtons, de zombies et de fantômes baveux ?

Source : https://pydefis.callicode.fr/defis/C22_VampireSurvivors/txt

Exercice Bonus 4.56 (Pydéfis – SW I : À l'assaut de Gunray. Découpage de la porte blindée)

La visite de Qui-Gon Jinn et Obi-Wan Kenobi sur le vaisseau de la fédération qui commande le blocus de la planète Naboo tourne mal et Qui-Gon, armé de son sabre laser doit percer la porte blindée derrière laquelle Nute Gunray s'est terré.

On modélise la découpe au sabre laser de la manière suivante : chaque seconde, le sabre perce le blindage sur une épaisseur E (exprimée en centimètres). Durant cette seconde, il produit un volume de métal en fusion égal à environ 8E (exprimé en centimètres-cubes).

À mesure que le métal fond le sabre perce moins vite. Si le volume de métal en fusion autour du sabre est V (exprimé en centimètres-cubes) à un instant donné, dans le seconde qui suit, le sabre pénétrera dans le blindage d'une épaisseur $E = 3 - 0.005V$

Exemple

- Lorsque le sabre commence à percer (à l'instant $t = 0$), il n'y a pas de métal en fusion, donc $V = 0$.
- En une seconde le sabre perce sur une profondeur de $E = 3 - 0.005V = 3$ cm. Le sabre a donc percé 3 cm en une seconde. Mais il a aussi produit une quantité de métal en fusion égale à 8E, c'est-à-dire 24 cm^3.
- Durant la deuxième seconde, puisque le volume de métal en fusion est maintenant 24 cm^3, le sabre va pénétrer de $3 - 0.005 \times 24 = 2.88$ cm supplémentaires. Au total il aura donc percé en 2 secondes $3 + 2.88 = 5.88$ cm. Mais durant cette deuxième seconde, il aura fait fondre $8 \times 2.88 = 23.04$ cm^3. Il y a donc maintenant (au bout de 2 secondes) $24 + 23.04 = 47.04$ cm^3 de métal en fusion.
- Durant la troisième seconde, il percera donc de $3 - 0.005 \times 47.04 = 2.7648$ cm. Au bout de trois secondes, le sabre aura pénétré au total de $5.88 + 2.7648 = 8.6448$ cm.

Défi : la porte blindée fait 70 cm d'épaisseur. Au bout de combien de secondes Qui Gon aura-t-il percé la moitié de la porte ? Et au bout de combien de secondes aura-t-il percé toute la porte ?

Source : https://pydefis.pydefis.callicode.fr/defis/PorteBlindeeSabre/txt

Exercice Bonus 4.57 (Pydéfis – L'hydre de Lerne)

Histoire Pour son deuxième travail, Eurysthée demanda à Hercule de tuer l'Hydre, une sorte de dragon possédant plusieurs têtes et qui hantait les marais de Lerne. Pour mener à bien sa mission, Hercule, muni de sa seule épée, décida de trancher les têtes de l'Hydre. La tâche n'était pas si facile et les têtes repoussaient parfois lorsqu'il les coupait. Toutefois, la repousse des têtes de l'Hydre suivait une règle simple, ainsi que la stratégie d'Hercule :

- À chaque coup d'épée, Hercule coupait la moitié des têtes restantes.
- Si après une coupe, il restait un nombre impair de têtes, alors le nombre de têtes restantes triplait instantanément, et une tête supplémentaire repoussait encore.
- Si à un moment l'Hydre ne possédait plus qu'une seule tête, Hercule pouvait l'achever d'un coup d'épée

supplémentaire.

Exemple

- Si l'Hydre a 6 têtes, Hercule en coupe la moitié au premier coup d'épée. Il en reste 3. Instantanément, le nombre de têtes triple et il en pousse une autre. Il y a maintenant 10 têtes.
- Au second coup d'épée, Hercule en coupe 5. Des têtes repoussent, il y en a maintenant 16.
- Au troisième coup d'épée, Hercule coupe 8 têtes. Il en reste 8. Rien ne repousse.
- Au quatrième coup d'épée, Hercule coupe 4 têtes, il en reste 4. Rien ne repousse.
- Au cinquième coup d'épée, Hercule coupe 2 têtes, il en reste 2. Rien ne repousse.
- Au sixième coup d'épée, Hercule coupe une tête. Il n'en reste plus qu'une.
- Le septième et dernier coup d'épée permet d'achever l'Hydre.

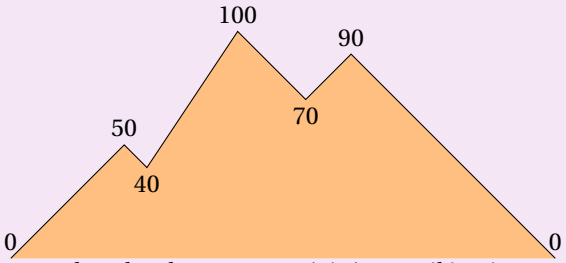
Si l'Hydre a 6 têtes, Hercule doit donc donner 7 coups d'épée pour la vaincre.

Défi : le nombre réel de têtes de l'Hydre est donné 8542. Combien de coups d'épée seront nécessaires pour venir à bout du monstre ?

Source : https://pydefis.callicode.fr/defis/Herculito02Hydre/txt

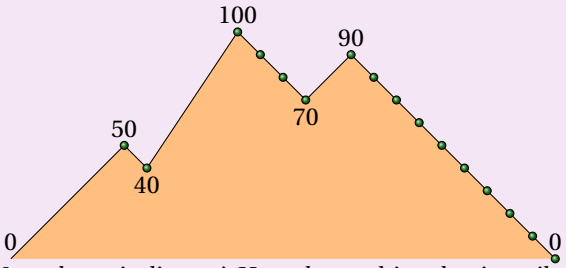
Exercice Bonus 4.58 (Pydéfis – Le sanglier d'Érymanthe)

Pour son quatrième travail, Eurysthée demanda à Hercule de capturer vivant le sanglier d'Érymanthe. Gigantesque, celui-ci dévastait avec rage le nord-ouest de l'Arcadie. Après avoir débusqué le sanglier, Hercule le poursuivit dans les montagnes en lui jetant des pierres. Le profil montagneux était assez accidenté et ressemblait à ceci :



Hercule, pour économiser ses forces, ne jetait des pierres que dans les descentes. Précisément, il jetait une pierre tous les 10 mètres (changement d'altitude). Ainsi, dans une descente de 30 mètres, il jetait 4 pierres.

On peut produire un relevé du profil des montagnes, en donnant les altitudes de chaque sommet et chaque col. Dans l'exemple qui précède, le relevé donnerait $0, 50, 40, 100, 70, 90, 0$. À partir de ce relevé uniquement, on peut voir qu'il y a 3 descentes, de 10, 30 et 90 mètres. Hercule jettera donc 16 pierres sur le sanglier.

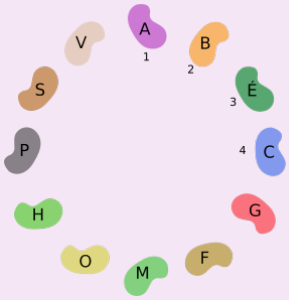


Défi : un relevé du profil réel vous est donné en entrée. Vous devez indiquer à Hercule combien de pierre il aura à jeter sur le sanglier.

Source du défi avec la vraie liste : https://pydefis.callicode.fr/defis/Herculito04Sanglier/txt

Exercice Bonus 4.59 (Pydéfis – Les dragées surprises)

Harry, Ron et Hermione sont dans le Poudlard express et Harry est heureux de partager ses dragées surprises de Bertie Crochue, achetées l'an passé chez Honeydukes. Il lui reste exactement 12 dragées : Aubergine, Bouillabaisse, Épinards, Chaussettes, Glace, Foie, Morve de Troll, Œuf Pourri, Herbe, Poubelle, Saucisse, et Vomi. Les partager va être difficile. Hermione propose de les disposer en cercle :



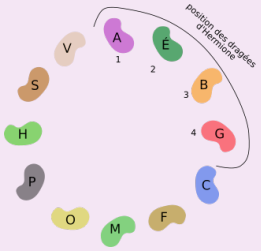
Puis, elle commence à compter : 1 sur Aubergine, 2 sur Bouillabaisse...

> Je vous propose de mélanger un peu tout ça. Je vais tourner en comptant, et toutes les 5 dragées, j'inverserai la dragée avec celle située juste avant :
>
> - 1, 2, 3, 4, 5 : j'échange Glace et Chaussette.
> - 1, 2, 3, 4, 5 : j'échange Poubelle et Herbe.

- 1, 2, 3, 4, 5 : j'échange Épinards et Bouillabaisse.
- etc.

Je vais faire ça pendant un moment, ce qui va mélanger les dragées. Ensuite, je prendrai les 4 premières (positions 1, 2, 3 et 4), Ron les 4 suivantes, et Harry les 4 dernières.

Voici la position des dragées après ces 3 premiers échanges :



Ron a l'air soupçonneux... : "Combien d'échanges tu vas faire exactement ?"

Hermione : "Oh, peu importe, je vais choisir un nombre au hasard, assez grand pour que ce soit bien mélangé."

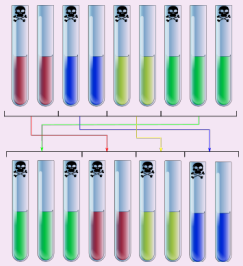
Sachant qu'Hermione ne raffole pas des parfums qui restent, exceptés Aubergine, Épinards, Glace et Herbe, qui semblent un peu moins mauvais que les autres, et qu'elle souhaite terminer cette histoire au plus vite, combien d'échanges doit-elle faire (inclus les 3 qu'elle a déjà faits lors de son explication) ?

Source : https://pydefis.callicode.fr/defis/DrageesBertie/txt

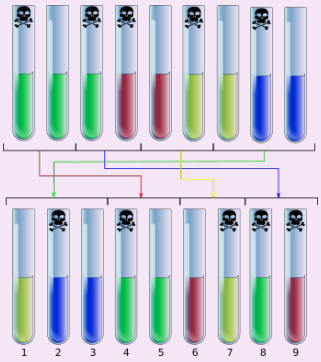
Exercice Bonus 4.60 (Pydéfis – Les tubes à essai d'Octopus. Méli-mélo de poison)

Toujours prêt à mettre au défi son ennemi Spiderman, le Dr Octopus lui présente une série de 9 tubes à essais dont ceux occupant les positions impaires sont empoisonnés.

« Je vais mélanger les tubes en suivant le protocole suivant : je sépare la série en 4 tas, contenant respectivement 2 (en rouge), 2 (en bleu), 2 (en jaune) et 3 (en vert) tubes ; les couleurs sont uniquement là pour mieux visualiser le fonctionnement. Puis, je place le dernier tas en premier (les verts), le premier tas en second (les rouges), le troisième tas (les jaunes) et enfin le second (les bleus). Ci-contre ce que cela donne. »



« Et je recommence. Je fais 4 tas de 2, 2, 2 et 3, puis je place le dernier tas en premier, etc. : »



« Je vais faire mon opération de mélange plusieurs fois aussi vite que mes bras mécaniques le permettent et tu vas devoir sélectionner les 4 tubes que tu boiras. Mouahahaha. Dans le cas qui précède, il faudrait que tu boives les tubes en position 1, 3, 5, et 6. »

« Mais attention... on ne va pas jouer avec 9 tubes de couleur marqués par une tête de mort. À la place, nous allons prendre 65 tubes incolores. Le premier, le troisième, le cinquième etc... seront empoisonnés. Au lieu de faire 4 tas de 2, 2, 2, et 3 tubes, je vais faire 4 tas de 14, 14, 14, et 23 tubes. Ainsi, si les tubes sont rangés dans l'ordre : 1, 2, 3,... 65, après une étape de mélange, les tubes seront dans l'ordre :

« 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28

« Je vais refaire cette opération de mélange plusieurs fois. À toi de me donner la liste des positions des 32 tubes que tu boiras. »

Exemple

- Si Octopus n'avait réalisé qu'un seul mélange, Spidey aurait dû boire les tubes occupant les positions : 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65
- Si Octopus avait réalisé en tout deux mélanges, Spidey aurait dû boire les tubes occupant les positions : 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 38, 40, 42, 44, 46, 48, 50, 53, 55, 57, 59, 62, 64

Défi : le Dr Octopus opère 48 mélanges. Aidez Spidey en lui indiquant la liste des tubes qu'il doit boire.

Source : https://pydefis.callicode.fr/defis/MelangeTubes/txt

Exercice Bonus 4.61 (Pydéfis – Le cours de potions)

Severus Rogue entre dans la salle du cours de potions :

> Prenez chacun quatre fioles vides de 25 centilitres. Dans la première, versez 20 centilitres de décoction d'Armoise, dans la seconde, versez 20 centilitres de décoction d'orties, et dans la troisième, versez 20 centilitres de décoction de fleur d'Aconit. Weasley, reposez immédiatement la quatrième fiole. À présent posez dans l'ordre les 4 fioles devant vous. La première est celle contenant l'Armoise et la dernière est vide. Vous allez verser le tiers du contenu de la première fiole dans la seconde, si c'est possible. S'il n'y a pas assez de place libre dans la seconde fiole pour accueillir le tiers du volume de la première, remplissez-la simplement à ras bord. Attention à ne rien renverser, Potter. Maintenant, versez un tiers de ce qui reste de la première fiole dans la troisième fiole. Et enfin, un tiers de ce qui reste de la première fiole dans la quatrième, qui est pour le moment vide. Bien. Vos 4 fioles contiennent maintenant respectivement 6.7 cl, 25 cl, 25 cl et 3.3 cl. Je suppose que vous avez compris ?
>
> À présent vous répétez l'opération, mais en versant le contenu de la deuxième fiole dans la troisième, la quatrième et la première. Vos 4 fioles contiennent maintenant respectivement 12.2 cl, 11.1 cl, 25 cl, et 11.7 cl. Recommencez encore, mais en vidant la fiole 3 (dans les fioles 4, 1 et 2), puis la fiole 4 (dans les fioles 1, 2 et 3). Recommencez à partir de la fiole 1. Lorsque chaque fiole aura été vidée 3 fois, combien contiendront chaque fiole ? Asseyez-vous, Miss Granger. Voyons... une idée Monsieur Potter ?

Pour valider le défi et sortir Harry de ce mauvais pas, indiquez la quantité de liquide contenue dans les quatre fioles, en centilitres. Par exemple, au bout de deux phases de vidage, il faudrait répondre 12.2, 11.1, 25, 11.7. Indiquez le volume contenu dans les fioles après la douzième opération de vidage (chaque fiole aura alors été vidée 3 fois).

Source : https://pydefis.callicode.fr/defis/CoursPotion1/txt

Exercice Bonus 4.62 (Pydéfis – Désamorçage d'un explosif (I))

Une bombe dévastatrice a été placée à Los Angeles par un membre des Maîtres du mal. Black Widow a réussi à la trouver et doit maintenant tenter de la désamorcer.

Une fois la bombe ouverte, c'est un véritable sac de nœuds. Il y a 1000 fils numérotés de 0 à 999, et il faut en couper un seul, le bon, pour arrêter le compte à rebours.

Heureusement, les Avengers ont pu fournir un manuel de désamorçage à Black Widow. Celui-ci indique (il est en russe, nous avons traduit pour vous) :

> Le numéro du fil à couper peut être déduit du numéro de série de la bombe ainsi :
>
> 1. commencez par relever le numéro de série
> 2. coupez le numéro de série en 2. Les trois premiers chiffres forment le nombre U et les 3 derniers le nombre N
> 3. répétez N fois les opérations 4 et 5 suivantes en partant du nombre U
> 4. multipliez ce nombre par 13
> 5. ne conservez que les 3 derniers chiffres.
>
> Une fois cet algorithme terminé, le nombre obtenu est le numéro du fil à couper.

Testez votre code : par exemple, si le numéro de série avait été 317010, il aurait fallu couper le fil 133.

Défi : indiquez à Black Widow le numéro du fil à couper pour valider le défi si le numéro de série est 449149.

Source : https://pydefis.callicode.fr/defis/Desamorcage01/txt

Exercice Bonus 4.63 (Pydéfis – Méli Mélo de nombres)

Soit $a = 195$ et $b = 117$. À partir d'un nombre de 4 chiffres, comme $u = 9697$, on fabrique un nouveau nombre avec la méthode suivante :

- tout d'abord, on sépare les 2 derniers chiffres des deux premiers, ce qui donne deux nombres : 96 et 97, qu'on ajoute ; nous obtenons 193.
- Puis, on multiplie ce résultat par a, et on ajoute b, ce qui donne 37752.
- Enfin, on calcule le reste de la division entière de 37752 par 9973, ce qui donne 7833.

Ainsi, à partir du nombre 9697, nous avons fabriqué le nouveau nombre 7833. On recommence cette opération n fois, ce qui construit une suite de nombres.

Éventuellement, un des nombres de la suite peut ne compter que 1, 2 ou 3 chiffres. L'opération est quand même possible. Pour calculer le nombre qui vient après 137, on sépare les deux derniers chiffres du nombre et on obtient les deux nombres 1 et 37 (attention, pas 13 et 7, mais 1 et 37), qu'on ajoute, etc. De même, si le nombre à transformer est 8, les deux nombres à ajouter seront 0 (le nombre de centaines), et 8 (le reste de la division par 100), etc.

Testez votre code : si $u = 3456$ et $n = 5$, il faut répondre 8641 car la suite de nombres calculés vaut 3456, 7694, 3348, 5939, 9254, 8641.

Défi : si $u = 3773$ et $n = 194$, quel nombre obtient-on si on applique la transformation ci-dessus n fois, en partant de u ?

Source : https://pydefis.callicode.fr/defis/Melange/txt

Exercice Bonus 4.64 (Pydéfis – Suite Tordue)

L'objet de cet exercice est de construire une suite de nombres en suivant certaines règles. Pour passer d'un nombre u au suivant il faut, après avoir écrit u sur 4 chiffres (en complétant éventuellement avec des 0 à gauche), ajouter le nombre formé des deux premiers chiffres de u avec le nombre formé des deux derniers chiffres de u, multiplier ce résultat par 191, et ajouter 161, prendre le reste de la division entière de ce nouveau résultat par 9973. Le nouveau nombre obtenu est le suivant dans la suite.

Testez votre code : par exemple, si u vaut 4267, on commence par calculer $42 + 67 = 109$. Puis on multiplie par 191 et on ajoute 161 pour trouver $109 \times 191 + 161 = 20980$. Enfin, on prend le reste de la division entière par 9973, ce qui nous donne 1034. Autre exemple, si u vaut 112, on commence par ajouter 01 et 12, pour trouver 13. Puis on multiplie par 191 et on ajoute 161 pour trouver 2644. Enfin, on prend le reste de la division entière par 9973, ce qui nous donne 2644.

Défi : l'entrée du problème est constituée de 2 valeurs : u_1 (premier terme de la suite) et n. Que vaut u_n ?

Source : https://pydefis.callicode.fr/defis/SuiteTordue/txt

Exercice Bonus 4.65 (Pydéfis – Bombe à désamorcer)

Afin de pouvoir enfin opérer sur le terrain, il vous reste à passer un examen pratique : le désamorçage de bombe.

Vous pouvez manipuler 5 fils : un noir, un rouge, un vert, un jaune, et un bleu. Sur ce type de bombe, le désamorçage consiste à débrancher les 5 fils, dans le bon ordre. L'essentiel du problème est donc de déterminer dans quel ordre il faut débrancher les fils. Chaque couleur correspond à un numéro : 1 pour le noir, 2 pour le rouge, 3 pour le vert, 4 pour le jaune et 5 pour le bleu.

La donnée de 5 chiffres indique l'ordre de coupure des fils. **Par défaut, il s'agit de 34125**, ce qui signifie qu'il faut couper en premier le vert (3), en deuxième le jaune (4), en troisième le noir (1), en quatrième le rouge (2) et enfin le bleu (5).

Avant que la bombe ne soit amorcée, la combinaison de désamorçage (34125) a toutefois été modifiée. On lui a fait subir des permutations. Une permutation consiste à échanger 2 chiffres de la combinaison. On décrit la permutation en donnant les positions des 2 chiffres à échanger. Par exemple, la permutation 14 signifie qu'il faut échanger le premier chiffre et le quatrième.

Voici un exemple (pour tester votre code) : la combinaison de départ est le code sortie d'usine 34125. Supposons qu'on ait appliqué les permutations 14, 25, 13. Le code devient alors

$$34125 \xrightarrow[\text{permutation 14}]{} 2413524135 \xrightarrow[\text{permutation 25}]{} 2513425134 \xrightarrow[\text{permutation 13}]{} 15234$$

Le résultat après les 3 permutations est 15234. Cela signifie qu'il faudra couper en premier le fil noir (1), en deuxième le fil bleu (5), en troisième le fil rouge (2), en quatrième le fil vert (3) et en dernier le fil jaune (4).

Dans quel ordre il faut couper les fils si la liste des permutations effectuées avant amorçage est la suivante ?

```
Perm = [41, 35, 13, 51, 34, 42, 23, 31, 13, 51, 32, 32, 43, 24, 54, 34, 34, 41, 35, 52, 15, 12, 43,
↪  52, 14, 24, 35, 13, 12, 31, 51, 31, 51, 35, 45, 15, 21, 42, 25, 32, 34, 21, 13, 12, 51, 13, 45,
↪  52, 14, 54, 34, 34, 42, 34, 21, 51, 54, 34, 51, 43, 31, 24, 31, 23, 51, 25, 23, 53, 12, 35, 41,
↪  31, 15, 35, 45, 24, 45, 12, 34, 45, 12, 12, 12, 15, 35, 51, 34, 12, 54, 32, 12, 25, 41, 45, 32,
↪  53, 35, 45, 41, 32]
```

Source : https://pydefis.callicode.fr/defis/Desamorcage03/txt

Exercice 4.66 (Cadeaux)

Le premier jour de l'an on reçoit un cadeau de 1 centime (0.01 €). Le lendemain on reçoit un cadeau de 2 centimes, ce qui nous fait une cagnotte de 3 centimes. Chaque jour suivant, le montant qu'on reçoit double par rapport à la veille. L'idée est de déterminer combien de jours cela prendra avant que le montant total des cadeaux reçus ne dépasse 1 million d'euros, et également de calculer le montant du dernier cadeau ainsi que le total des cadeaux reçus.

Correction

On initialise les variables `cadeau_total` (le montant total reçu), `cadeau_du_jour` (le montant reçu chaque jour, initialement 0.01 €), et jour (le jour actuel, démarrant à 1). Ensuite, on utilise une boucle `while` pour augmenter jour et `cadeau_total` tant que le montant total reçu est inférieur à 1 million d'euros. À chaque itération, le montant du cadeau quotidien (`cadeau_du_jour`) double et est ajouté au montant total.

```
cadeau_total, cadeau_du_jour, jour = .01, .01, 1
print(f'''Le jour {jour} je reçois {cadeau_du_jour}€ ainsi au totale j'ai {cadeau_total:.2f}€.''')

while cadeau_total < 1.e6:
    jour += 1
    cadeau_du_jour *= 2
    cadeau_total += cadeau_du_jour

print(f'''Le jour {jour} je reçois {cadeau_du_jour}€ ainsi au totale j'ai {cadeau_total:.2f}€.''')
```

```
Le jour 1 je reçois 0.01€ ainsi au totale j'ai 0.01€.
Le jour 27 je reçois 671088.64€ ainsi au totale j'ai 1342177.27€.
```

Exercice 4.67 (Affichage)

Pour $n \in \mathbb{N}$ donné, calculer $\mathrm{N} = \sum_{i=1}^{n} i$, puis afficher "$1+2+\cdots+n = \mathrm{N}$". Par exemple, si $n = 3$, on doit afficher la chaîne "1+2+3=6".

Correction

$$1+2+3+4+5+6+7+8+9+10 = 55$$

```
n = 10
for i in range(1,n):
    print(i," + ", end=" ")
print(n,"=",int(n*(n+1)/2))
```

Exercice Bonus 4.68 (Pydéfis – Insaisissable matrice)

On considère la matrice suivante

$$\begin{pmatrix} 36 & 19 & 27 & 36 & 7 & 10 \\ 2 & 18 & 3 & 33 & 2 & 21 \\ 26 & 27 & 4 & 22 & 30 & 31 \\ 29 & 36 & 7 & 20 & 6 & 30 \\ 30 & 6 & 14 & 23 & 15 & 13 \\ 22 & 10 & 10 & 35 & 15 & 22 \end{pmatrix}$$

Cette matrice va évoluer au cours du temps, et le contenu m_{ij} d'une case est transformé, à chaque étape, en $(11m_{ij}+4)\%37$ où $a\%b$ donne le reste de la division entière de a par b. À chaque étape de calcul, tous les nombres de la matrice sont simultanément modifiés.

Défi : que vaut la somme des valeurs contenues dans la matrice après application de 23 étapes ?

Source : https://pydefis.callicode.fr/defis/AlgoMat/txt

Exercice 4.69 (Maximum d'une liste de nombres sans la fonction `max`)

Soit une liste de nombres. Trouver le plus grand et le plus petit élément de cette liste **sans utiliser les fonctions prédéfinies** `max`, `min`, `sorted` ni la méthode `sort`.

Correction

```
L = [10, 5, 15, -2, 17, -22]
mymax = L[0]
mymin = L[0]
for ell in L:
    if ell>mymax:
        mymax = ell
    if ell<mymin:
        mymin = ell
print(f"L={L}, mymax(L)={mymax}, mymin(L)={mymin}")
```

```
L=[10, 5, 15, -2, 17, -22], mymax(L)=17, mymin(L)=-22
```

Exercice 4.70 (Chaîne la plus longue d'une liste de strings)

Soit une liste de chaînes de caractères (toutes de longueurs différentes). Trouver la plus longue et la plus courte chaîne de cette liste **sans utiliser les fonctions prédéfinies** `max`, `min`, `sorted` ni la méthode `sort`.

Correction

```
L = ["Je", "Vous", "Bonjour", "Adieu"]
mymax = L[0]
mymin = L[0]
for ell in L:
    if len(ell)>len(mymax):
        mymax = ell
    if len(ell)<len(mymin):
        mymin = ell
print(f"{L = }, mymax(L) = {mymax}, mymin(L) = {mymin}")
```

```
L = ['Je', 'Vous', 'Bonjour', 'Adieu'], mymax(L) = Bonjour, mymin(L) = Je
```

Exercice Bonus 4.71 (Défi Turing n°9 – triplets pythagoriciens)

Le triplet d'entiers naturels non nuls (a,b,c) est pythagoricien si $a^2+b^2=c^2$. Par exemple, $(3,4,5)$ est un triplet pythagoricien.

Parmi les triplets pythagoriciens (a,b,c) tels que $a+b+c=3600$, donner le produit $a\times b\times c$ le plus grand.

Correction

Nous n'avons pas besoin de faire varier les 3 valeurs, en faire varier 2 suffit car si nous connaissons, par exemple, la valeur de c et a, nous pouvons en déduire celle de b en résolvant l'équation $a+b+c=p$ où p est le périmètre (ici $p=3600$).

De plus, nous savons que $a<c$, $b<c$ et nous pouvons étudier seulement les cas $a<b$. Dans ce cas, comme $a<b<c$ et $a+b+c=p$, alors $3a<p<3c$.

```
l = []
p = 3600
for a in range(1,int(p/3)+1): # au lieu de range(1,p-1) car a<b<c implique 3a<a+b+c=p
    for c in range(int(p/3),p): # au lieu de range(a,p-a) car a<b<c implique p=a+b+c<3c
        b = p-a-c
        if (a*a+b*b)==(c*c):
            l.append(a*b*c)
print(max(l))
```

```
1654329600
```

Exercice Bonus 4.72 (Défi Turing n°11 - nombre miroir)
On appellera "miroir d'un nombre n" le nombre n écrit de droite à gauche. Par exemple, le miroir de 7423 est 3247. Quel est le plus grand nombre inférieur à 10 millions ayant la propriété : "miroir de $n = 4n$" ?

Correction

```
l = []
for n in range(10**7):
    miroir = int(str(n)[::-1])
    if miroir==4*n:
        l.append(n)
print(max(l))
```

```
2199978
```

Exercice Bonus 4.73 (Défis Turing n°13, n°43 et n°73 – Carrés parfaits palindromes)
Un entier naturel est appelé *palindrome* lorsqu'il s'écrit de la même manière de gauche à droite et de droite à gauche (voir l'exercice 2.14). Par exemple, 235532, 636 ainsi que tous les nombres à un seul chiffre sont des palindromes. Dans cet exercice, on s'intéresse aux *carrés parfaits palindromes*, c'est-à-dire aux nombres de la forme n^2 qui sont des palindromes en écriture décimale.

Défi Turing n°13 : On sait que le plus petit carré palindrome possédant un nombre *pair* de chiffres est $698896 = 836^2$. Déterminer le carré palindrome suivant ayant un nombre pair de chiffres, ainsi que l'entier dont il est le carré.

Défi Turing n°43 : Calculer la somme de tous les entiers naturels n tels que n^2 soit un palindrome comportant au plus 13 chiffres.

Défi Turing n°73 : Déterminer le plus grand palindrome à 7 chiffres dont le carré est également un palindrome.

Correction
Un entier est palindrome si et seulement si son écriture décimale se lit de la même manière de gauche à droite et de droite à gauche. Une méthode simple pour vérifier cette propriété consiste à transformer le nombre en chaîne de caractères et à comparer cette chaîne avec son inverse.

Dans les trois défis, on parcourt un ensemble fini d'entiers naturels, dont les bornes sont déterminées à l'aide d'un raisonnement mathématique préalable.

Défi Turing n°13 On cherche le carré palindrome à nombre pair de chiffres *suivant* 836^2. On part donc de $n = 837$ et on teste successivement les entiers jusqu'à trouver la première solution. L'instruction `break` permet d'interrompre immédiatement la boucle dès que la solution est trouvée.

```
n = 837
while True:
    carre = n * n
    carre_str = str(carre)
    if len(carre_str) % 2 == 0 and carre_str == carre_str[::-1]:
        print(f"n = {n} car n^2 = {carre}")
        break
    n += 1
```

```
n = 798644 car n^2 = 637832238736
```

Défi Turing n°43 Si n^2 comporte au plus 13 chiffres, alors $n^2 < 10^{13}$, donc $n < \sqrt{10^{13/2}}$. Il suffit donc de tester les entiers naturels strictement inférieurs à cette borne.

```
borne_n = int(10**(13/2)) + 1
somme_entiers = 0
for n in range(borne_n):
    carre = n*n
    carre_str = str(carre)
    if carre_str == carre_str[::-1]:
        somme_entiers += n
print(somme_entiers)
```

```
27974694
```

Défi Turing n°73 On cherche le *plus grand* palindrome à 7 chiffres dont le carré est aussi un palindrome. Un palindrome à 7 chiffres vérifie $10^6 \leq n < 10^8$. On parcourt donc ces entiers dans l'*ordre décroissant* et on s'arrête dès qu'une solution est trouvée.

```
borne_sup = 10**7 - 1
borne_inf = 10**6 - 1
for palindrome in range(borne_sup, borne_inf, -1):
    palindrome_str = str(palindrome)
    if palindrome_str == palindrome_str[::-1]:
        carre = palindrome * palindrome
        carre_str = str(carre)
        if carre_str == carre_str[::-1]:
            print(palindrome, carre)
            break
```

```
2001002 4004009004004
```

Exercice 4.74 (Entier palindrome dans une base *b*)

Déterminer si le nombre entier donné est un palindrome ou non en base B. Par exemple,

- 6 s'écrit 110 en base 2 ($6 = 0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2$) donc la réponse est `False`
- 34 s'écrit 114 en base 5 ($34 = 4 \times 5^0 + 1 \times 5^1 + 1 \times 5^2$) donc la réponse est `False`
- 455 s'écrit 111000111 en base 2 donc la réponse est `True`

Correction

```
# d en base 10 -> b en base B
for d,B in [ (455,2) , (3148,16) , (1442,10) ]:
    n = d
    b = []
    while n > 0:
        n, res = divmod(n,B)
        b.append(res)
    b = ''.join([str(c) for c in b] )[::-1]
    print( f"Le nombre {d} en base 10, s'écrit {b} en base {B}. Est-il palindrome? {b==b[::-1]}" )
```

```
Le nombre 455 en base 10, s'écrit 111000111 en base 2. Est-il palindrome? True
Le nombre 3148 en base 10, s'écrit 21421 en base 16. Est-il palindrome? False
Le nombre 1442 en base 10, s'écrit 1442 en base 10. Est-il palindrome? False
```

Exercice Bonus 4.75 (Défi Turing 22 – Les anagrammes octuples)

Mathilde a trouvé deux nombres de six chiffres étonnants. Lorsqu'on les multiplie par 8, on obtient un nombre de six chiffres qui s'écrit avec les mêmes chiffres rangés dans un ordre différent. Quels sont les nombres de Mathilde ?

Correction

```
for n in range(100000,int(1000000/8)+1):
    if sorted(str(n))==sorted(str(8*n)):
        print(n)
```

```
113967
116397
```

Exercice Bonus 4.76 (Défi Turing n°21 – Bonne année 2013 !)

2013 a une particularité intéressante : c'est la première année depuis 1987 à être composée de chiffres tous différents. Une période de 26 ans sépare ces deux dates.

Entre l'an 1 et 2013 (compris) :

1) Combien y a-t-il eu d'années composées de chiffres tous différents ? (les années de l'an 1 à l'an 9 seront comptées dans ce nombre).
2) Quelle a été la durée (en années) de la plus longue période séparant deux dates ayant des chiffres tous différents ?

Donner le produit des résultats de 1) et 2).

Correction

```
L = []
for n in range(1,2014):
    stringa = str(n)
    if len(stringa)==len(set(stringa)):
        L.append(n)

Q1 = len(L)
ecarts = [ L[i]-L[i-1] for i in range(1,len(L)) ]
Q2 = max(ecarts)
print(f"{Q1 = }, {Q2 = }, {Q1*Q2 = }")
```

```
Q1 = 1243, Q2 = 105, Q1*Q2 = 130515
```

Exercice Bonus 4.77 (Pydéfis – L'escargot courageux)

Un escargot veut gravir une tour de 324 mètres. Le premier jour, il monte de x centimètres. La première nuit, il glisse (vers le bas) de y centimètres. Chaque jour supplémentaire, il monte de 1 centimètre(s) de moins que la journée précédente. En revanche, la nuit il glisse toujours de y centimètres.

Dans la région où est située cette tour, il pleut toutes les 8 nuits. Lorsque ça se produit, au bout de la nuit, l'escargot se retrouve au même endroit que 48 heures auparavant.

En appelant le jour 0 celui de son départ (il part le matin), et sachant qu'il vient de pleuvoir la nuit dernière, quel jour l'escargot arrive-t-il en haut de la tour si $x = 1370$ et $y = 280$?

Pour vérifier la compréhension de l'énoncé, voici le début du parcours de l'escargot, pour $x = 1330$ et $y = 300$

```
Fin du jour    0 : altitude=1330 cm
Fin de la nuit 0 : altitude=1030 cm
Fin du jour    1 : altitude=2359 cm
Fin de la nuit 1 : altitude=2059 cm
Fin du jour    2 : altitude=3387 cm
Fin de la nuit 2 : altitude=3087 cm
Fin du jour    3 : altitude=4414 cm
Fin de la nuit 3 : altitude=4114 cm
Fin du jour    4 : altitude=5440 cm
Fin de la nuit 4 : altitude=5140 cm
Fin du jour    5 : altitude=6465 cm
Fin de la nuit 5 : altitude=6165 cm
Fin du jour    6 : altitude=7489 cm
Fin de la nuit 6 : altitude=7189 cm
Fin du jour    7 : altitude=8512 cm
Fin de la nuit 7 : altitude=6165 cm
Fin du jour    8 : altitude=7487 cm
Fin de la nuit 8 : altitude=7187 cm
Fin du jour    9 : altitude=8508 cm
Fin de la nuit 9 : altitude=8208 cm
```

Source : https://pydefis.callicode.fr/defis/Escargot/txt

Exercice Bonus 4.78 (Pydéfis – Mon beau miroir...)

On appelle l'image miroir d'un nombre, le nombre lu à l'envers. Par exemple, l'image miroir de 324 est 423. Un nombre est un palindrome s'il est égal à son image miroir. Par exemple, 52325, ou 6446 sont des palindromes. À partir d'un nombre de départ, nous pouvons l'ajouter à son image miroir, afin d'obtenir un nouveau nombre, puis recommencer avec ce nouveau nombre jusqu'à obtenir un palindrome. À ce nombre de départ correspondent ainsi 2 valeurs : le palindrome obtenu, ainsi que le nombre d'addition qu'il a fallu faire pour l'obtenir. Par exemple, pour le nombre de départ 475, nous obtenons :

`475 + 574 = 1049` ⇝ `1049 + 9401 = 10450` ⇝ `10450 + 5401 = 15851`

Le denier nombre, 15851, est un palindrome. Pour le nombre de départ 475, nous atteignons donc le palindrome 15851 en 3 étapes.

Dans cet exercice, l'entrée est une séquence de nombres. Vous devez répondre en donnant une séquence de couples : le palindrome obtenu, et le nombre d'étapes. Par exemple, si l'entrée est `(844, 970, 395, 287)` vous devrez obtenir `[[7337, 3] , [15851, 3] , [881188, 7] , [233332, 7]]`

Qu'obtient-on avec la séquence d'entrée : `746, 157, 382, 461, 885, 638, 462, 390, 581, 692` ?

Source : https://pydefis.callicode.fr/defis/MiroirAjout/txt

Exercice Bonus 4.79 (Pydéfis – Persistance)

Il s'agit ici d'étudier les «suites de persistance». Ces suites sont obtenues, à partir de n'importe quel nombre entier, en calculant le produit de ses chiffres, et en recommençant. La suite de persistance de l'entier 347, par exemple est

$$347 \to 3 \times 4 \times 7 = 84 \to 8 \times 4 = 32 \to 3 \times 2 = 6$$

On s'arrête lorsqu'il ne reste plus qu'un chiffre.

On cherche à savoir quels sont les chiffres sur lesquels on tombera le plus souvent (ici, nous sommes tombés sur le chiffre 6). 0 est exclu de cette étude, car il est obtenu en écrasante majorité (dès qu'il y a un 0 dans un nombre, la suite s'arrête sur 0).

Indiquer combien de fois chaque chiffre entre 1 et 9 a été obtenu comme terminaison de la suite de persistance, pour tous les entiers entre L = 1701 et R = 4581.

Par exemple, si les nombres donnés étaient L = 371 et R = 379, il faudrait répondre avec la liste $[0,2,0,1,0,3,0,2,0]$. En effet, si on construit toutes les suites de persistance

$371 \to 21 \to 2$	$372 \to 42 \to 8$	$373 \to 63 \to 18 \to 8$
$374 \to 84 \to 32 \to 6$	$375 \to 105 \to 0$	$376 \to 126 \to 12 \to 2$
$377 \to 147 \to 28 \to 16 \to 6$	$378 \to 168 \to 48 \to 32 \to 6$	$379 \to 189 \to 72 \to 14 \to 4$

On obtient donc le chiffre 1 zéro fois, le chiffre 2 deux fois, le chiffre 3 zéro fois, le chiffre 4 une fois… le chiffre 8 deux fois et le chiffre 9 zéro fois. La réponse est en conséquence $0,2,0,1,0,3,0,2,0$

Source : https://pydefis.callicode.fr/defis/Persistance/txt

Exercice Bonus 4.80 (Pydéfis – Toc Boum)

Défi : dans cet exercice, un nombre entier n vous est donné en entrée. Ce nombre peut s'écrire : $n = 13a + 7b$ où a et b sont des entiers strictement positifs. Si plusieurs couples a, b conviennent, il faut trouver le couple tel que a et b soient des nombres les plus proches possibles.

Testez votre code : si l'entrée fournie est 287, les couples a, b possibles sont $(7,28)$ $(14,15)$ et $(21,2)$. Le couple a, b solution (celui pour lequel a et b sont les plus proche) est donc $(14,15)$.

Source : https://pydefis.callicode.fr/defis/TocBoum/txt

Exercice Bonus 4.81 (Pydéfis – Les juments de Diomède)

Histoire : pour son huitième travail, Eurysthée demanda à Hercule de lui ramener les juments de Diomède. Ces quatre féroces animaux se nourrissaient de chair humaine et Diomède, un des fils d'Arès, les nourrissait avec les voyageurs de passage.

Hercule se rendit donc en Thrace et entreprit de calmer la faim des juments afin de les capturer. N'ayant jamais eu l'intention de sacrifier ses amis ou les innocents de passage, il avait pris soin de faire embarquer un grand nombre de paquets de croquettes pour chat sur son bateau (Hercule voyageait à pied, car il souffrait du mal de mer, mais son équipe voyageait en bateau).

Défi : sachant que chacun des quatre animaux, pour être repu, consommait 131 kg de croquettes et qu'Hercule possédait à son bord 20 sacs de 7 kg, 20 sacs de 11 kg et 20 sacs de 13 kg, combien de sacs de 7, 11 et 13 kg devrait-il débarquer pour apporter très exactement la quantité de nourriture nécessaire aux juments, ni moins (elle ne seraient pas repues), ni plus (ne pas pouvoir finir leur assiette mettait les juments particulièrement en colère) ? Parmi toutes les solutions possibles, Hercule voulait débarquer le moins de sacs. Et parmi les solutions qui satisfaisaient ce critère, il devait essayer, pour épargner ses compagnons, de débarquer le moins de sacs de 13 kg.

Testez votre code : si Hercule avait eu à son bord 7 sacs de 5 et 7 sacs de 9 kg, et si les juments avaient chacune consommé 23 kg, alors Hercule aurait dû débarquer 12 sacs : 1 sac de 5 kg, 6 sacs de 7 kg et 5 sacs de 9 kg. En effet, le total fait bien $1 \times 5 + 6 \times 7 + 5 \times 9 = 92$ kg $= 4 \times 23$ kg. La solution à ce problème serait donc 1, 6, 5. Remarquez que la solution 5, 7, 2 ne convient pas car elle nécessite plus de sacs (14 sacs au lieu de 12). La solution 2, 4, 6 ne convient pas non plus, car même si elle ne nécessite aussi que 12 sacs, il faut débarquer 6 gros sacs (au lieu de 5 pour la solution valide). Enfin, la solution 0, 8, 4 ne convient pas non plus, car on n'a que 7 sacs de chaque sorte, et non 8.

Source : https://pydefis.callicode.fr/defis/Herculito08Juments/txt

Exercice Bonus 4.82 (Pydéfis – Produit et somme palindromiques)

Nous cherchons ici les nombres entiers a, b, c, d tels que le produit $abcd$ et la somme $a+b+c+d$ soient des palindromes.

Par exemple :

- si $a = 15$, $b = 71$, $c = 59$, $d = 87$, le produit $abcd = 5466645$ est un palindrome ainsi que la somme $a+b+c+d = 232$;
- si $a = 13$, $b = 47$, $c = 98$, $d = 68$, le produit $abcd = 4071704$ est un palindrome, ce qui n'est pas le cas de la somme $a+b+c+d = 226$;

- si $a = 12$, $b = 4$, $c = 68$, $d = 37$, le produit $abcd = 120768$ n'est pas un palindrome, alors que la somme $a+b+c+d = 121$ l'est.

Défi : on donne en entrée les bornes `mini = 25` et `maxi = 95` (incluses) de a, b, c et d. Trouvez combien de quadruplets a, b, c, d sont tels que $abcd$ et $a+b+c+d$ soient tous les deux des palindromes.

Testez votre code : si `mini = 10` et `maxi = 28`, alors il y a 5 solutions :

$$(11,11,11,11) \to 11 \times 11 \times 11 \times 11 = 14641 \text{ et } 11+11+11+11 = 44$$
$$(11,11,19,25) \to 11 \times 11 \times 19 \times 25 = 57457 \text{ et } 11+11+19+25 = 66$$
$$(13,13,14,26) \to 13 \times 13 \times 14 \times 26 = 61516 \text{ et } 13+13+14+26 = 66$$
$$(14,14,14,24) \to 14 \times 14 \times 14 \times 24 = 65856 \text{ et } 14+14+14+24 = 66$$
$$(17,21,22,28) \to 17 \times 21 \times 22 \times 28 = 219912 \text{ et } 17+21+22+28 = 88$$

Source : https://pydefis.callicode.fr/defis/Palindromes/txt

CHAPITRE 5

Définitions en compréhension

Le concept fondamental des **expressions en compréhension** est de permettre d'écrire du code de manière concise, lisible et proche du raisonnement mathématique.

5.1. Listes en compréhension

La compréhension de liste est un moyen concis et expressif de créer des listes en Python. Sa syntaxe est très proche de la notation mathématique utilisée pour définir des ensembles :

Ensemble mathématique :	$\{$	$f(x)$	$\mid$	x	$\in$	E	$\}$
Liste en compréhension :	[	f(x)	for	x	in	E	]

Syntaxe d'une liste en compréhension

```
[ function(item) for item in list if condition(item) ]
```

Comparons ces deux scripts, qui créent une liste contenant les carrés des entiers de 0 à 4 : $[0,1,4,9,16]$. La version avec compréhension est non seulement plus concise, mais aussi plus performante.

① **Avec une boucle classique :**

```
L = []
for i in range(5):
    L.append(i**2)
print(f"{L = }")
L = [0, 1, 4, 9, 16]
```

② **Avec une liste en compréhension :**

```
L = [i**2 for i in range(5)]
print(f"{L = }")

L = [0, 1, 4, 9, 16]
```

On peut ajouter un filtre

① **Avec une boucle classique :**

```
L = []
for i in range(5):
    if i**0.5<2:
        L.append(i**2)
print(f"{L = }")
L = [0, 1, 4, 9]
```

② **Avec une liste en compréhension :**

```
L = [i**2 for i in range(5) if i**0.5<2]
print(f"{L = }")
L = [0, 1, 4, 9]
```

Autres exemples

1. Génrer une liste de tuples

```
E = [2, 4, 8, 10]
L = [(x, x**3) for x in E]

print(f"{E = }")
print(f"{L = }")
```

```
E = [2, 4, 8, 10]
L = [(2, 8), (4, 64), (8, 512), (10, 1000)]
```

2. Boucles imbriquées.

```
E = ["a", "b"]
F = range(3)
L = [x*y for x in E for y in F]

print(f"{E = }")
print(f"{F = }")
print(f"{L = }")
```

```
E = ['a', 'b']
F = range(0, 3)
L = ['', 'a', 'aa', '', 'b', 'bb']
```

3. Conditions multiples.

```
E  = [1, 2, 3, 4, 5, 6, 7]
L1 = [x**2 for x in E]
L2 = [x for x in E if x%2 == 0]
L3 = [x**2 for x in E if x**2%2 == 0]
L4 = [x for x in [a**2 for a in E] if x%2 == 0]

print(f"{E  = }")
print(f"{L1 = }")
print(f"{L2 = }")
print(f"{L3 = }")
print(f"{L4 = }")
```

```
E  = [1, 2, 3, 4, 5, 6, 7]
L1 = [1, 4, 9, 16, 25, 36, 49]
L2 = [2, 4, 6]
L3 = [4, 16, 36]
L4 = [4, 16, 36]
```

4. Condition ternaire.

```
L = [x+1 if x >= 3 else x+5 for x in range(6)]
print(f"{L = }")
```

```
L = [5, 6, 7, 4, 5, 6]
```

5. Transposée d'une matrice.

```
M = [[1, 2, 3], [4, 5, 6]]
M_transpose = [[row[i] for row in M] for i in range(len(M[0]))]
print(f"{M_transpose = }")
```

```
M_transpose = [[1, 4], [2, 5], [3, 6]]
```

Version alternative (liste de tuples) :

```
M = [[1, 2, 3], [4, 5, 6]]
M_transpose = list(zip(*M))
print(f"{M_transpose = }")
```

```
M_transpose = [(1, 4), (2, 5), (3, 6)]
```

5.2. ☆ Dictionnaires en compréhension

Comme pour les listes, il est possible de construire des **dictionnaires en compréhension**. Cela permet de créer un nouveau dictionnaire à partir d'un itérable ou d'un autre dictionnaire, tout en appliquant des transformations ou des filtres sur les clés et/ou les valeurs.

Syntaxe d'un dictionnaire en compréhension

```
{ nouvelle_clé : nouvelle_valeur for ancienne_clé,
ancienne_valeur in dictionnaire.items() if condition }
```

La structure est donc similaire à celle d'une liste en compréhension, mais avec deux expressions séparées par un `:`, la première pour la clé, la seconde pour la valeur.

1. Doubler les valeurs.

```
D1 = {'a':1, 'b':2, 'c':3, 'd':4}
D2 = {k: v*2 for k, v in D1.items()}

print(f"{D1 = }")
print(f"{D2 = }")
```

```
D1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
D2 = {'a': 2, 'b': 4, 'c': 6, 'd': 8}
```

2. Modifier les clés.

```
D1 = {'a':1, 'b':2, 'c':3, 'd':4}
D3 = {k*2: v for k, v in D1.items()}

print(f"{D1 = }")
print(f"{D3 = }")
```

```
D1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
D3 = {'aa': 1, 'bb': 2, 'cc': 3, 'dd': 4}
```

3. Ajouter une condition.

```
D1 = {'a':1, 'b':2, 'c':3, 'd':4}
D4 = {k: v for k, v in D1.items() if v%2 == 0}

print(f"{D1 = }")
print(f"{D4 = }")
```

```
D1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
D4 = {'b': 2, 'd': 4}
```

4. Créer un dictionnaire à partir d'une liste.

```
E = [1, 2, 3, 4]
carres = {x: x**2 for x in E}

print(f"{E = }")
print(f"{carres = }")
```

```
E = [1, 2, 3, 4]
carres = {1: 1, 2: 4, 3: 9, 4: 16}
```

5.3. ☆ Ensembles en compréhension

Les ensembles peuvent eux aussi être construits en compréhension :

Syntaxe d'un ensemble en compréhension

$$\{\ \texttt{function(item) for item in } \overbrace{\texttt{list if condition(item)}}^{\texttt{E}}\ \}$$

Par exemple, l'instruction suivante génère un ensemble contenant seulement deux éléments, car les doublons sont automatiquement éliminés :

```
>>> {skill for skill in ['SQL', 'SQL', 'PYTHON', 'PYTHON']}
{'PYTHON', 'SQL'}
```

On peut également ajouter des conditions :

```
>>> {skill for skill in ['GIT', 'PYTHON', 'SQL'] if skill not in {'GIT', 'PYTHON', 'JAVA'}}
{'SQL'}
```

5.4. Exercices

Rappel de la syntaxe générale pour définir une liste en compréhension

Soit L une séquence (liste, chaîne de caractères, tuple ou itérateur). Grâce aux compréhensions de liste, on peut créer une nouvelle liste en parcourant L de différentes manières : par élément, par indice, ou avec la fonction `enumerate` pour accéder simultanément aux indices et aux éléments.

```
# --- Accès par élément ---
[ f(x) for x in L if g(x) ]
[ f(x,y) for x,y in zip(L,M) if g(x,y) ]      # avec deux listes

# --- Accès par indice ---
[ f(L[i]) for i in range(len(L)) if g(L[i]) ]
[ f(L[i],M[i]) for i in range(min(len(L),len(M))) if g(L[i],M[i]) ]

# --- Obtenir les indices ---
[ i for i in range(len(L)) if g(L[i]) ]
[ i for i in range(min(len(L),len(M))) if g(L[i],M[i]) ]

# --- Indices et éléments avec enumerate ---
[ i for i,x in enumerate(L) if g(x) ]
[ i for i,(x,y) in enumerate(zip(L,M)) if g(x,y) ]
```

- **Accès par élément** : on parcourt directement chaque élément de la séquence, ce qui est simple et lisible.
- **Accès par indice** : on utilise les indices pour manipuler les éléments, utile quand on a besoin de l'indice ou qu'on travaille sur plusieurs listes simultanément.
- **Indices avec `enumerate`** : combine les deux approches, donnant à la fois l'indice et l'élément à chaque itération, ce qui est souvent pratique pour des transformations ou des filtrages conditionnels.

Exercice 5.1 (Sous-listes)

Soit L une liste d'entiers, par exemple `L = [1, 3, 6, 2, 3, 4, 9, 19]`. À partir de L, construisez en compréhension les sous-listes suivantes :

- `P_val`, qui contient uniquement les éléments pairs de L,
- `P_idx`, qui contient les indices des éléments pairs de L,
- `P_pos`, qui contient les éléments de L situés à des positions paires.

Correction

```
L = [1, 3, 6, 2, 3, 4, 9, 19] # Exemple de liste
P_val = [x for x in L if x % 2 == 0] # Liste des éléments pairs
# Alternative : P_val = [L[i] for i in range(len(L)) if L[i] % 2 == 0]
P_idx = [i for i in range(len(L)) if L[i] % 2 == 0] # Indices des éléments pairs
# Alternative : P_idx = [i for (i, x) in enumerate(L) if x % 2 == 0]
# Remarque : ne pas utiliser P_idx = [L.index(x) for x in L if x % 2 == 0] car cela échouerait si L
↪  contient des doublons : L.index(x) renvoie l'indice de la PREMIÈRE occurrence de x dans L
P_pos = L[::2] # Éléments en positions paires
print(f"{L = }, {P_val = }, {P_idx = }, {P_pos = }")
```

```
L = [1, 3, 6, 2, 3, 4, 9, 19], P_val = [6, 2, 4], P_idx = [2, 3, 5], P_pos = [1, 6, 3, 9]
```

Remarque : la fonction `enumerate` est utile pour obtenir simultanément l'indice `i` et la valeur `x` lors de l'itération.

Exercice 5.2 (Somme des carrés)

Soit L une liste de nombres. Construire en compréhensions la liste des carrés des éléments de L. En calculer ensuite la somme. Par exemple, si `L=[0,1,2]`, on doit obtenir s=5.

Correction

```
L = list(range(3)) # exemple
C = [x**2 for x in L]
s = sum(C)
print(f"{L = }, {C = }, {s = }")
```

```
L = [0, 1, 2], C = [0, 1, 4], s = 5
```

Exercice 5.3 (Œufs)

Chaque semaine, Jean ramasse entre 40 et 200 œufs qu'il va vendre au marché. Ce soir, veille de marché, il est perplexe : s'il met ses œufs dans des emballages de 6, il en reste 2 ; s'il utilise des emballages de 10, il en reste encore 2. Il me faudrait, dit-il, des emballages de 8 pour tout contenir exactement. Combien d'œufs a-t-il ramassés ?

Correction

Source : https://fr.wikibooks.org/wiki/Math%C3%A9matiques_avec_Python_et_Ruby

Si, regroupées par 10, il en reste 2, cela signifie que le nombre cherché a 2 unités. On peut donc directement générer une suite de candidats avec `range(42, 193, 10)`.

```
# L = [ n for n in range(40,201) if n%6==2 and n%10==2 and n%8==0]
# L = [ n for n in range(40,201,8) if n%6==2 and n%10==2]
L = [ n for n in range(42,193,10) if n%6==2 and n%8==0]
sol = L[0]
print(sol)
# Vérification
print(*[f"{sol}%{i} = {sol%i}" for i in (6,8,10)], sep=", ")
```

```
152
152%6 = 2, 152%8 = 0, 152%10 = 2
```

Exercice 5.4 (Rallye mathématique de la Réunion 2005, exercice 2)

Pour organiser une grande manifestation sportive, le professeur d'éducation physique doit rassembler sur le stade un important groupe d'élèves. Le nombre d'élèves est compris entre 2800 et 2900. Il en profite pour leur faire remarquer que, regroupés par 2, puis par 3, puis par 4, puis par 5, puis par 6, il en reste toujours 1 ; mais, ô miracle, en se regroupant par 7, il ne reste personne. Combien d'élèves a-t-il rassemblés ?

Correction

Si, regroupés par 2, il reste 1 élève, cela signifie que le nombre recherché est impair. Par ailleurs, s'il ne reste aucun élève lorsqu'on regroupe par 7, le nombre recherché est un multiple de 7. On peut donc directement générer une suite de candidats avec `range(2807, 2901, 14)`, qui contient les nombres impairs multiples de 7 dans l'intervalle donné. Ensuite on impose les autres conditions, en commençant de la plus restrictive.

```
L = [ n for n in range(2807,2901,14) if n%6 == 1 and n%5 == 1 and n%4 == 1 and n%3 == 1 ]
```

Notons que la condition "regroupés par 6 il en reste toujours 1" contient les conditions "regroupés par 2, puis par 3, il en reste toujours 1" et il n'est donc pas nécessaire de les coder :

```
L = [ n for n in range(2807,2901,14) if n%6 == 1 and n%5 == 1 and n%4 == 1 ]
```

On peut toutefois éviter de coder ces trois conditions en observant que le plus petit commun multiple (PPCM) de 4, 5 et 6 est 60. Donc n doit vérifier $n \equiv 1 \pmod{60}$.

```
L = [ n for n in range(2807,2901,14) if n%60==1]
sol = L[0]
print(sol)
# Vérification à posteriori
print(*[f"{sol}%{i} = {sol%i}" for i in range(2,8)], sep=", ")
```

```
2821
2821%2 = 1, 2821%3 = 1, 2821%4 = 1, 2821%5 = 1, 2821%6 = 1, 2821%7 = 0
```

Exercice Bonus 5.5 (Pydéfi – Le jardin des Hespérides)

Histoire : les Hespérides, filles d'Atlas, habitaient un merveilleux jardin dont les pommiers donnaient des pommes en or. Pour son 11e travail, Eurysthée demanda à Hercule de ramener ces pommes. Une fois atteint le jardin merveilleux, l'oracle Nérée apprit à Hercule qu'il pourrait repartir avec une partie des pommes... à condition qu'il montre ses facultés en calcul mental. Nérée lui tint ce propos :

> J'ai empilé les pommes d'or pour toi, sous la forme d'une pyramide. L'étage le plus haut ne contient

qu'une pomme. L'étage juste en dessous forme un carré 2×2 (contenant 4 pommes), l'étage juste en dessous forme un carré 3×3 (contenant 9 pommes). La pyramide que tu vois contient 50 étages. L'étage de base contient donc 2500 pommes... Je suis d'accord pour te laisser partir avec les pommes contenues dans certains étages. Précisément, si un étage contient un nombre de pommes multiple de 3, tu peux l'emporter. Si tu m'annonces combien de pommes tu emporteras au total, je te laisserai partir avec les pommes...

Défi : vous devez aider Hercule en lui indiquant le nombre de pommes qu'il pourra emporter pour une pyramide de 50 étages.

Testez votre code : par exemple, si la pyramide n'avait compté que 6 étages, chaque étage aurait été composé de : 1, 4, 9, 16, 25 et 36 pommes. Hercule aurait pu emporter les 9 pommes de l'étage 3 (car 9 est un multiple de 3) et les 36 pommes de l'étage 6 (car 36 est un multiple de 3). Au total il aurait donc emporté 45 pommes.

Source : https://pydefis.callicode.fr/defis/Herculito11Pommes/txt

Exercice 5.6 (Pièces magiques - bis)

Créer la même liste de l'exercice 4.6 en une seule ligne de code.

Correction

C'est une suite arithmétique de raison $r = 67$, d'où $S_n = 20 + 67n$.

```
S = [20 + 67 * semaine for semaine in range(53)]
print(f"S[0]={S[0]}, S[1]={S[1]}, S[52]={S[52]}")
```

```
S[0]=20, S[1]=87, S[52]=3504
```

Exercice 5.7 (Temperature moyenne)

Soit L une liste de 7 sous-listes correspondant aux températures mesurées au cours de la semaine (la première liste contient les mesures de lundi, etc.). Calculer les jours les plus chauds. Si plusieurs jours ont la même température moyenne, renvoyer tous ces jours.

Suggestion : utiliser les fonctions `sum`, `len`, `max` et `index`.

Correction

Pour calculer la somme des éléments de la liste, on utilise la fonction `sum()` ; pour calculer combien d'éléments contient la liste, on utilise la fonction `len()` ; pour trouver les jours avec la température moyenne la plus élevée, on utilise les fonctions `max()` et une boucle pour vérifier chaque jour.

```
# Liste des jours de la semaine
jours = ["Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche"]

# Liste des températures mesurées au cours de la semaine
L = [
    [15, 16, 15, 14],  # Lundi
    [17, 18, 19, 20],  # Mardi
    [21, 22, 23, 24],  # Mercredi
    [25, 26, 27, 28],  # Jeudi
    [29, 30, 31, 32],  # Vendredi
    [33, 34, 35, 36],  # Samedi
    [37, 38, 39, 40]   # Dimanche
]


moyennes = [sum(jour) / len(jour) for jour in L]
jour_max_temp = max(moyennes)
jours_chauds = [i for i, moyenne in enumerate(moyennes) if moyenne == jour_max_temp]

print(f"Les jours les plus chauds sont : {[jours[i] for i in jours_chauds]} avec une température moyenne
↪  de {jour_max_temp:.2f}°C.")
```

```
Les jours les plus chauds sont : ['Dimanche'] avec une température moyenne de 38.50°C.
```

Exercice 5.8 (Liste de `str` ⇝ liste de `int`)

Soit `S` une liste de chaînes de caractères, chacune constituée uniquement de chiffres. Construire par compréhension la liste `L` qui contient les nombres entiers associés à chaque chaîne. Par exemple, si `S = ["5", "10", "-15"]`, alors `L = [5, 10, -15]`.

Correction

```
S = ["5", "10", "-15"] # exemple
L = [int(x) for x in S]
print(f"{L = }, {S = }")
```

```
L = [5, 10, -15], S = ['5', '10', '-15']
```

Exercice 5.9 (Chaîne de caractères ⇝ liste de `str`)

Soit `s` une chaîne de caractères. Construire en compréhension la liste qui contient chaque caractère. Exemple : si `s="Ciao"`, on doit obtenir la liste `["C", "i", "a", "o"]`.

Correction

```
>>> s = "Ciao" # exemple
>>> L = [c for c in s]
>>> print(L)
['C', 'i', 'a', 'o']
```

ou, plus simplement

```
>>> s = "Ciao"
>>> L = list(s)
>>> print(L)
['C', 'i', 'a', 'o']
```

Exercice 5.10 (Somme des chiffres d'un nombre)

Soit $n \in \mathbb{N}$. Générer, par compréhension, une liste qui contient chaque chiffre de n (chaque élément de cette liste doit être un `int`). Calculer ensuite la somme de ces éléments. Par exemple, si `n=30071966`, on doit obtenir 32.

Bonus : et si $n \in \mathbb{R}$?

Correction

À l'exercice 2.11 nous avons vu comment calculer le nombre de chiffres qui composent un nombre $n \in \mathbb{N}$, simplement en transformant le nombre en chaîne de caractères puis en comptant la longueur de cette chaîne. Dans cet exercice, on peut s'en inspirer. On transforme le nombre n en une chaîne de caractères (`str(n)`). On lit les chiffres un par un (`for c in str(n)`) et on les transforme en entier (`int(c)`). Enfin on additionne les éléments de cette liste :

```
n = 30071966 # exemple
s = sum([int(c) for c in str(n)])
print(f"La somme des chiffres de {n} est {s}")
```

```
La somme des chiffres de 30071966 est 32
```

Pour $n \in \mathbb{R}$ il ne faut garder que les chiffres (pas l'éventuel "-" ou "."):

```
n = -3007.1966 # exemple
s = sum([int(c) for c in str(n) if c.isdigit()])
print(f"La somme des chiffres de {n} est {s}")
```

```
La somme des chiffres de -3007.1966 est 32
```

Remarque : avec des nombres très grands, cette méthode est inefficace. Pire, si le nombre a plus de 4300 chiffres, comme par exemple 7^{1000}, on ne peut pas faire une conversion en chaîne de caractères. On peut cependant obtenir le même résultat en divisant n par 10 pour récupérer ses chiffres.

```
n = 30071966
s = 0
while n > 0:
    n, c = divmod(n,10)
    s += c
print(s)
```

```
32
```

Exercice 5.11 (Défi Turing n°5 – somme des chiffres d'un nombre)

$2^{15} = 32768$ et la somme de ses chiffres vaut $3+2+7+6+8 = 26$. Que vaut la somme des chiffres composant le nombre 2^{2222} ?

Correction

Pour résoudre ce problème, on trouve d'abord la valeur de 2^{2222}, on convertit ce nombre en chaîne de caractères, on lis les chiffres un par un, on les convertit en entier, enfin on les additionne :

```
>>> print(sum([int(x) for x in str(2**15)]))
26
```

```
>>> print(sum([int(x) for x in str(2**2222)]))
2830
```

Exercice 5.12 (Liste de Moyennes)

Soit P une liste de listes de nombres. Définir par compréhension une liste qui contient les moyennes arithmétiques des sous-listes de P.

Par exemple, si `P = [ [1,2,3], [4,5,6,7], [5,-1,8], [10,11] ]`, on doit obtenir [2.0, 5.5, 4.0, 10.5].

Correction

```
>>> P = [ [1,2,3], [4,5,6,7], [5,-1,8], [10,11] ] # exemple
>>> [ sum(l)/len(l) for l in P ]
[2.0, 5.5, 4.0, 10.5]
```

Exercice 5.13 (Diviser une liste en sous-listes)

Diviser une liste L en plusieurs sous-listes d'une taille donnée `size`. Si la taille de L n'est pas un multiple de `size`, la dernière sous-liste contiendra les éléments restants. Exemple : pour `L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` et `size = 4`, le résultat attendu est la liste (qui contient des sous-listes) `[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10]]`.

Correction

```
>>> L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> size = 4
>>> [ L[i:i+size] for i in range(0, len(L), size) ]
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10]]
```

L'expression `L[i:i+size]` extrait une tranche de L. Si `i+size` dépasse la longueur de L, Python retourne automatiquement les éléments restants grâce à la gestion flexible des indices dans les tranches.

Exercice 5.14 (`argmin` : position du minimum d'une liste de nombres)

Pour une liste L de nombres, l'argmin correspond à l'indice du plus petit élément (s'il y en a plusieurs, on retient le premier indice). Écrire une instruction qui calcule cet indice.

Suggestion : utiliser la fonction `enumerate()` pour créer une liste de tuples. Comment agit la fonction `min()` sur une liste de tuples ?

Correction

Solution en une ligne : `min([ (value,idx) for idx,value in enumerate(L) ])[1]`

Explications : la fonction `enumerate(L)` associe à chaque élément de la liste son indice et renvoie des tuples de la forme `(indice, valeur)`. Lorsque la fonction `min()` est appliquée sur une liste de tuples, la comparaison se fait *lexicographiquement* : on compare d'abord la première composante, puis, en cas d'égalité, la deuxième. Ainsi, le tuple renvoyé correspond à la plus petite valeur de la liste et, en cas d'égalité, au premier indice où elle apparaît. On extrait alors la deuxième composante de ce tuple.

```
L = [10, -22, 5, 15, -2, 17, -22] # exemple
print(f"{L = }")

tmp = [(v, i) for i, v in enumerate(L)]
print(f"{tmp = }")
```

```
min_tmp = min(tmp)
print(f"{min_tmp = }")

print(f"Le minimum vaut {min_tmp[0]} et sa position est {min_tmp[1]}")
```

```
L = [10, -22, 5, 15, -2, 17, -22]
tmp = [(10, 0), (-22, 1), (5, 2), (15, 3), (-2, 4), (17, 5), (-22, 6)]
min_tmp = (-22, 1)
Le minimum vaut -22 et sa position est 1
```

Variante: on peut aussi laisser les tuples sous la forme `(indice, valeur)` et indiquer explicitement à `min()` de comparer selon la deuxième composante et non la première (qui est le critère par défaut) à l'aide du paramètre `key`:

```
min_idx, min_val = min(enumerate(L), key=lambda x: x[1])
print(f"Le minimum vaut {min_val} et sa position est {min_idx}")
```

```
Le minimum vaut -22 et sa position est 1
```

Exercice 5.15 (Chaturanga)

Selon la légende, l'ancêtre des échecs, le Chaturanga, aurait été créé en Inde par le sage Sissa. Le roi de l'époque fut tellement séduit par ce nouveau jeu qu'il offrit au sage de choisir tout ce qu'il désirait en récompense. Cependant, Sissa ne demanda rien et resta silencieux. Le roi, offensé par son attitude, le pressa de s'exprimer, mais le sage, blessé, décida de se venger. Il demanda au roi de déposer un grain de riz sur la première case de l'échiquier, deux grains sur la deuxième case, quatre grains sur la troisième case et ainsi de suite jusqu'à la dernière case. Le roi accepta, mais le lendemain matin, il fut alerté par son intendant que la quantité de riz requise était beaucoup trop grande pour être satisfaite. Pourquoi une telle affirmation ?

1. Générer par compréhension la liste du nombre de grains sur chacune des cases.
2. Déterminer le nombre total de grains sur l'échiquier.
3. Chercher la masse d'un grain de riz et déterminer la masse totale de riz sur l'échiquier.

Correction

```
L = [2**i for i in range(64)]

# Quelque prints
for idx in [0,1,2,10,63]:
    print(f"Grains de riz sur la case {idx+1} = {L[idx]}")

nb_grain = sum(L)
print(f"Grains de riz sur l'échiquier = {nb_grain} i.e. {nb_grain:g} ")

un_grain = 0.02 # en gramme
print(f"Un grain de Riz a une masse de {un_grain}g")

tot_gr = un_grain*nb_grain
tot_kg = tot_gr*10**(-3)
tot_Tonne = tot_gr*10**(-6)
print(f"Masse sur l'échiquier {tot_gr:g}g = {tot_kg:g}Kg = {tot_Tonne:g}t ")
print(f"La masse de la Terre est d'environ {5.972e21}t \n")
```

```
Grains de riz sur la case 1 = 1
Grains de riz sur la case 2 = 2
Grains de riz sur la case 3 = 4
Grains de riz sur la case 11 = 1024
Grains de riz sur la case 64 = 9223372036854775808
Grains de riz sur l'échiquier = 18446744073709551615 i.e. 1.84467e+19
Un grain de Riz a une masse de 0.02g
Masse sur l'échiquier 3.68935e+17g = 3.68935e+14Kg = 3.68935e+11t
La masse de la Terre est d'environ 5.972e+21t
```

Nous pouvons calculer directement le nombre de grains sur l'échiquier. En effet, notons b_n le nombre de grains de blé sur la case n, n allant de 0 à 63. La suite (g_n) est géométrique de raison 2 car $g_{n+1} = 2g_n$ donc $g_n = 2^n g_0 = 2^n$. Ainsi la

somme totale des grains de blé sera

$$\sum_{n=0}^{63} g_n = \sum_{n=0}^{63} 2^n = \frac{1-2^{64}}{1-2} = 2^{64}-1 = 18446744073709551615 \simeq 18\cdot 10^{18}.$$

Au lieu de comparer la masse de riz avec la masse de la Terre, essayons d'avoir une idée du volume occupé par cette quantité. Un grain de riz est, grosso modo, un cylindre de diamètre 1 mm et de hauteur 5 mm. Ainsi on pourrait faire tenir 200 grains de riz dans un centimètre cube (= 1000 mm^3). On peut commencer nos calculs. Si l'on peut faire tenir 200 grains de riz dans un centimètre cube, alors il nous en faut $200\cdot 100^3$ pour un mètre cube et $200\cdot 100^3\cdot 1000^3$ pour un kilomètre cube. Si on divise cette quantité de grains par $2\cdot 10^{17}$ on obtient le volume de total de notre montagne rizière : 92 kilomètres cubes. Ce volume est tout aussi difficile à imaginer. En remarquant que la France a une superficie d'environ 375000 km^2, on peut se représenter la quantité de riz demandée par l'inventeur du jeu d'échecs de la manière suivante : avec elle on pourrait couvrir toute la France d'une couche de riz de 13 centimètres de haut (car 13 centimètres correspondent à peu près à 7.5 millièmes de kilomètre et $675000/7500 = 90$).

Exercice 5.16 (Filtrer une liste)

Soit la liste

```
ma_liste_de_mots = [ "maths", "info", "python", "exposants", "alpha", "fonctions",
                     "parabole", "equilateral", "orthogonal", "cercles", "isocèle" ]
```

1. Créer et afficher une liste qui contient uniquement les mots commençant par une voyelle.
2. Créer et afficher une liste qui contient uniquement les mots qui se terminent par un `"s"`.
3. Créer et afficher une liste qui contient uniquement les mots de longueur strictement inférieure à 7.
4. Créer et afficher une liste qui contient uniquement les mots commençant par une voyelle et dont la longueur strictement inférieure à 7.

Correction

`mot[0]` récupère la première lettre de `mot`. Pour vérifier que c'est une voyelle, on vérifie qu'elle appartient à `"aàeéèiouùy"`. `mot[-1]` récupère la dernière lettre de `mot`. `len(mot)` récupère la longueur de `mot`.

```
print( [ mot for mot in ma_liste_de_mots if mot[0] in "aàeéèiouùy" ] )
print( [ mot for mot in ma_liste_de_mots if mot[-1]=="s" ] )
print( [ mot for mot in ma_liste_de_mots if len(mot)<7 ] )
print( [ mot for mot in ma_liste_de_mots if (mot[0] in "aàeéèiouùy") and (len(mot)<7) ] )
```

```
['info', 'exposants', 'alpha', 'equilateral', 'orthogonal', 'isocèle']
['maths', 'exposants', 'fonctions', 'cercles']
['maths', 'info', 'python', 'alpha']
['info', 'alpha']
```

Exercice 5.17 (Liste des diviseurs)

Pour un entier $n \in \mathbb{N}$ donné, calculer la liste de ses **diviseurs propres** (c'est-à-dire les diviseurs de n qui sont strictement inférieurs à n). On rappelle que d divise n si et seulement si `n%d==0`.

Correction

On utilise une compréhension de liste pour générer la liste des diviseurs propres de n. On parcourt les entiers d de 1 à $n-1$ inclus, et sélectionne uniquement ceux pour lesquels $n \mod d = 0$ (ce qui signifie que d divise n). Cela génère la liste des diviseurs propres de n.

```
>>> n = 100
>>> [d for d in range(1,n) if (n%d==0)]
[1, 2, 4, 5, 10, 20, 25, 50]
```

Amélioration : au lieu de parcourir les entiers de 1 à $n-1$, on parcourt seulement ceux de 1 à $\lfloor \frac{n}{2} \rfloor$ (en arrondissant à l'entier supérieur). Cela suffit car si $d > n/2$, alors $n/d < 2$ et ne peut donc pas être un diviseur propre.

```
>>> n = 100
>>> [d for d in range(1,n//2+1) if (n%d==0)]
[1, 2, 4, 5, 10, 20, 25, 50]
```

Exercice Bonus 5.18 (Défi Turing n°18 – Somme de nombres non abondants)

Un nombre parfait est un nombre dont la somme de ses diviseurs propres est exactement égal au nombre. Par exemple, la somme des diviseurs propres de 28 serait $1+2+4+7+14=28$, ce qui signifie que 28 est un nombre parfait. Un nombre n est appelé déficient si la somme de ses diviseurs propres est inférieur à n et on l'appelle abondant si cette somme est supérieure à n. Comme 12 est le plus petit nombre abondant ($1+2+3+4+6=16$), le plus petit nombre qui peut être écrit comme la somme de deux nombres abondants est 24.

Trouver la somme de tous les entiers positifs inférieurs ou égaux à 2013 qui **ne peuvent pas** être écrits comme la somme de deux nombres abondants.

Correction

```
# (dictionnaire) nb : liste diviseurs
div = { x : [ i for i in range(1,x) if x%i==0 ]  for x in range(2,2014) }
# (dictionnaire) nb : somme ses diviseurs propres
s = { k : sum(v)  for k,v in div.items() }
# (liste) nb abondant
a = [ k for k,v in s.items() if v>k ]
# (ensemble) nb somme de 2 nb abondnats (sans doublons)
L = set([a1+a2 for a1 in a for a2 in a])
print(sum([ x for x in range(2014) if x not in L]))
```

```
577167
```

Exercice Bonus 5.19 (Défis Turing n°71 – ensembles)

On remarque que $567^2 = 321489$ utilise tous les chiffres de 1 à 9 exactement une fois (si on exclut le carré lui-même). Quel est le seul autre nombre vérifiant cette propriété ?

Correction

On cherche un nombre à trois chiffres dont le carré, combiné à ses propres chiffres, forme une suite contenant exactement les chiffres de 1 à 9, sans doublon ni zéro.

Une première approche consiste à exploiter les **ensembles** pour éliminer automatiquement les doublons.

```
>>> [i  for i in range(100,1000) if len( set(str(i)+str(i*i)).difference(set("0")) )==9]
[567, 854]
```

Une autre méthode, sans utiliser d'ensembles, consiste à **trier les chiffres** du nombre et de son carré, puis à comparer le résultat à la liste des chiffres de 1 à 9.

```
>>> [n for n in range(100,1000) if (sorted(str(n)+str(n**2))==list('123456789'))]
[567, 854]
```

Exercice Bonus 5.20 (Somme et de différence de deux carrés)

Le nombre 40 est une somme de deux carrés puisque $40 = 2^2 + 6^2$. Il est aussi la différence de deux carrés puisque $40 = 7^2 - 3^2$. On cherche tous les entiers inférieurs ou égales à 100 ayant cette propriété.

Calculer tous les entiers inférieurs ou égales à 100 pouvant être exprimés comme la somme de deux carrés.

Calculer ensuite tous les entiers positifs ou nul résultant de la différence de deux carrés.

Enfin, trouver les valeurs communes entre ces deux ensembles.

Correction

```
A = {i ** 2 + j ** 2 for i in range(10) for j in range(10) if i**2 + j ** 2 <= 100}
B = {i ** 2 - j ** 2 for i in range(10) for j in range(10) if i**2 - j ** 2 >= 0}
C = A & B
print(f"{C = }")
```

```
C = {0, 1, 4, 5, 8, 9, 13, 16, 17, 20, 25, 32, 36, 40, 45, 49, 64, 65, 72, 80, 81}
```

Exercice 5.21 (Soustraire deux listes)

Soit deux listes de nombres $A = [a_i]_{i=0}^n$ et $B = [b_i]_{i=0}^n$ de même longueur. Construire la liste $D = [d_i]_{i=0}^n$ telle que $d_i = a_i - b_i$ pour tout i.

Correction

On peut parcourir deux listes simultanément de deux manières: soit directement par éléments grâce à `zip`, soit en utilisant les indices.

```
A = [ 1, 4,  6, 19, 125]
B = [78, 2, 46, 19,  56]
print( [ a - b for a, b in zip(A, B) ] )
print( [ A[i]-B[i] for i in range(len(A)) ] )
```

```
[-77, 2, -40, 0, 69]
[-77, 2, -40, 0, 69]
```

Exercice 5.22 (Jours du mois)

Soient les listes suivantes:

```
J = [ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ]
M = ['Janvier', 'Fevrier', 'Mars', 'Avril', 'Mai', 'Juin',
     'Juillet', 'Aout', 'Septembre', 'Octobre', 'Novembre', 'Decembre']
```

Générer en compréhension la **liste de tuples** `[ ('Janvier',31) , ('Fevrier',28) ... ]`

Correction

On peut parcourir deux listes simultanément de deux manières: soit directement par éléments grâce à `zip`, soit en utilisant les indices.

```
print( [ (M[i],J[i]) for i in range(len(J)) ] )
print( list(zip(M,J)) )
```

```
[('Janvier', 31), ('Fevrier', 28), ('Mars', 31), ('Avril', 30), ('Mai', 31), ('Juin', 30), ('Juillet',
↪  31), ('Aout', 31), ('Septembre', 30), ('Octobre', 31), ('Novembre', 30), ('Decembre', 31)]
[('Janvier', 31), ('Fevrier', 28), ('Mars', 31), ('Avril', 30), ('Mai', 31), ('Juin', 30), ('Juillet',
↪  31), ('Aout', 31), ('Septembre', 30), ('Octobre', 31), ('Novembre', 30), ('Decembre', 31)]
```

☆ Un dictionnaire est peut-être une structure un peu plus adaptée qu'une liste de tuples:

```
print( { m : j  for m,j in zip(M,J) } )
print( { M[i] : J[i]  for i in range(len(J)) } )
```

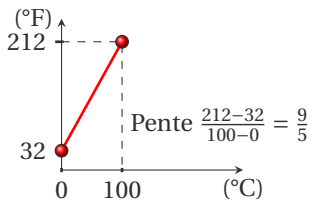
'Janvier' : 31, 'Fevrier' : 28, 'Mars' : 31, 'Avril' : 30, 'Mai' : 31, 'Juin' : 30, 'Juillet' : 31, 'Aout' : 31, 'Septembre' : 30, 'Octobre' : 31, 'Novembre' : 30, 'Decembre' : 31 'Janvier' : 31, 'Fevrier' : 28, 'Mars' : 31, 'Avril' : 30, 'Mai' : 31, 'Juin' : 30, 'Juillet' : 31, 'Aout' : 31, 'Septembre' : 30, 'Octobre' : 31, 'Novembre' : 30, 'Decembre' : 31

Exercice 5.23 (Conversion de températures)

Une température de 0°C correspond à 32°F tandis que 100°C correspondent à 212°F. La relation pour effectuer la conversion d'une valeur x de la température en (°C) vers l'unité (°F) est affine. Créez une liste de tuples où la première composante contient la valeur en Celsius (de 0°C à 100°C par pas de 10°C) et la deuxième l'équivalente en Fahrenheit.

Correction

La formule permettant la conversion d'une valeur numérique x de la température en °C vers l'unité °F est affine:



$$y = \frac{9}{5}x + 32$$

```
print([ (x, 9*x/5+32) for x in range(0,101,10) ])
```

[(0, 32.0), (10, 50.0), (20, 68.0), (30, 86.0), (40, 104.0), (50, 122.0), (60, 140.0), (70, 158.0), (80, 176.0), (90, 194.0), (100, 212.0)]

Exercice 5.24 (Années bissextiles)

Depuis l'ajustement du calendrier grégorien, l'année sera bissextile si l'année est

$$\Big(\text{divisible par 4} \quad \text{et} \quad \text{non divisible par 100}\Big) \qquad \text{ou} \qquad \Big(\text{divisible par 400.}\Big)$$

Ainsi,

- 2019 n'est pas bissextile car non divisible par 4 ni par 400
- 2008 était bissextile suivant la première règle (divisible par 4 et non divisible par 100)
- 1900 n'était pas bissextile car divisible par 4, mais aussi par 100 (première règle non respectée) et non divisible par 400 (seconde règle non respectée).
- 2000 était bissextile car divisible par 400.

Écrire la liste des années bissextiles entre l'année 1800 et l'année 2099 et la liste des années non bissextiles entre l'année 1800 et l'année 2000.

Correction

Ce programme traduit la définition des années bissextiles : "b est bissextile si et seulement si ($r = 0$ et $s \neq 0$) ou ($t = 0$)" où r, s et t désignent successivement les restes dans la division euclidienne de b par 4, 100 et 400. Attention aux parenthèses : "($r = 0$ et $s \neq 0$) ou ($t = 0$)" équivaut à "($r = 0$ ou $t = 0$) et ($r = 0$ ou $s \neq 0$)".

```
print([b for b in range(1800,2100) if (b%4==0 and b%100!=0) or (b%400==0)])
```

[1804, 1808, 1812, 1816, 1820, 1824, 1828, 1832, 1836, 1840, 1844, 1848, 1852, 1856, 1860, 1864, 1868, 1872, 1876, 1880, 1884, 1888, 1892, 1896, 1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936, 1940, 1944, 1948, 1952, 1956, 1960, 1964, 1968, 1972, 1976, 1980, 1984, 1988, 1992, 1996, 2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028, 2032, 2036, 2040, 2044, 2048, 2052, 2056, 2060, 2064, 2068, 2072, 2076, 2080, 2084, 2088, 2092, 2096]

La négation s'écrit : "n n'est pas bissextile si et seulement si ($r \neq 0$ ou $s = 0$) et ($t \neq 0$)"

```
print([n for n in range(1800,2000) if (n%4!=0 or n%100==0) and (n%400!=0)])
```

Si on a un doute pour la négation, on peut tout simplement écrire `if not((b%4==0 and b%100!=0) or (b%400==0))`.

Exercice Bonus 5.25 (Tri personnalisé)

Considérez une liste `x` d'entiers. Vous devez créer une nouvelle liste `y` qui contient les mêmes entiers triés de manière à ce qu'ils soient ordonnés comme s'ils étaient des nombres à cinq chiffres composés des trois premiers et des deux derniers chiffres de la valeur, indépendamment des chiffres du milieu.

Par exemple, si le tableau est [166552, 12389245, 88234, 74123245], le résultat serait trié comme s'il s'agissait des nombres [16652, 12345, 88234, 74145], et donc la liste triée serait : [12389245, 166552, 74123245, 88234].

Tous les nombres sont des entiers supérieurs ou égaux à 10000, sans zéros initiaux.

Notez qu'il est possible de spécifier une clé de comparaison qui sera utilisée pour effectuer le tri en utilisant l'option `key` de la fonction sorted.

Correction

Démarche :

1. Chaque entier est converti en une chaîne de caractères.
2. Les trois premiers et les deux derniers caractères de chaque entier sont extraits.
3. Les chaînes obtenues sont ensuite converties en entiers.
4. La liste est triée en fonction de ces entiers. La fonction lambda dans l'option `key` est utilisée pour indiquer à `sorted()` de comparer les éléments de la liste en fonction des valeurs dans `my_key`.

```
x = [166552, 166551, 12389245, 12389235, 88234, 74123245]

my_key = [int(str(num)[:3] + str(num)[-2:]) for num in x ]
sorted_indices = sorted(range(len(x)), key=lambda i: my_key[i])
y = [x[i] for i in sorted_indices]

print(f"{x = }")
print(f"{y = }")
```

```
x = [166552, 166551, 12389245, 12389235, 88234, 74123245]
y = [12389235, 12389245, 166551, 166552, 74123245, 88234]
```

Une autre idée consiste à créer une liste de tuple sachant que la fonction `sorted()` utilise la première composante d'un tuple pour les ordonner :

```
x = [166552, 166551, 12389245, 12389235, 88234, 74123245]

xx = [ ( int(str(num)[:3] + str(num)[-2:]) , num ) for num in x ]
sorted_xx = sorted(xx)
y = [x[1] for x in sorted_xx]

print(f"{x = }")
print(f"{y = }")
```

```
x = [166552, 166551, 12389245, 12389235, 88234, 74123245]
y = [12389235, 12389245, 166551, 166552, 74123245, 88234]
```

Exercice 5.26 (Vacances)

On réalise une enquête auprès de N français pour connaître l'endroit de leur dernier séjour ainsi que sa durée (on ne considère qu'un seul séjour par personne). Chaque participant est enregistré dans un tuple contenant deux valeurs entières : la première représente la durée du séjour en jours, et la seconde est le code du pays selon les conventions suivantes :

- personne n'étant pas partie en vacances : code 0,
- personne ayant passé des vacances en France : code 1,
- personne ayant voyagé à l'étranger : code > 1 (par exemple, Italie code 2, Espagne code 3, etc.).

Ces tuples sont rassemblés dans une liste.

Écrire un script qui crée les deux sous-liste "français partis en vacances en France", "français partis en vacances à l'étranger". Puis calculer et afficher :

1. le nombre de français ayant répondu à l'enquête,
2. le nombre de français partis en vacances en France et la durée moyenne de leur séjour,
3. le nombre de français partis en vacances à l'étranger et la durée moyenne de leur séjour.

Pour valider le script, créer un jeu de données pertinent.

Correction

```
L = [ (0,0) , (7,1) , (7,2) , (28,2) , (3,1) ]

# Sous-listes
no_vac = [ (j,p) for j,p in L if p==0]
oui_vac_fr = [ (j,p) for j,p in L if p==1]
oui_vac_et = [ (j,p) for j,p in L if p>1]

# Affichage
print(f"Nombre de français ayant répondu à l'enquête: {len(L)}")
print(f"Nombre de français partis en vacances en France: {len(oui_vac_fr)}. Durée moyenne de leur
↪  séjour: { sum([j for j,p in oui_vac_fr])/len(oui_vac_fr) }")
print(f"Nombre de français partis en vacances à l'étranger: {len(oui_vac_et)}. Durée moyenne de leur
↪  séjour: { sum([j for j,p in oui_vac_et])/len(oui_vac_et) }")
```

```
Nombre de français ayant répondu à l'enquête: 5
Nombre de français partis en vacances en France: 2. Durée moyenne de leur séjour: 5.0
Nombre de français partis en vacances à l'étranger: 2. Durée moyenne de leur séjour: 17.5
```

Exercice 5.27 (Produit matriciel)

Nous avons vu à l'exercice 2.35 qu'on peut représenter une matrice avec des listes de listes. En calcul matriciel, si $\mathbb{A}$ est une matrice de n lignes et p colonnes et $\mathbb{B}$ une matrice de p lignes et q colonnes, la matrice $\mathbb{C} = \mathbb{A}\mathbb{B}$ est une matrice de n lignes et q colonnes dont les éléments sont définis par

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}.$$

Écrire une boucle qui, pour $\mathbb{A}$ et $\mathbb{B}$ données, calcule $\mathbb{C}$.

Testez votre code sur les trois exemples suivants :

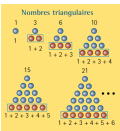
$$\overbrace{\begin{pmatrix}3&1&5\\2&7&0\end{pmatrix}}^{2\times 3}\overbrace{\begin{pmatrix}2&1&-1&0\\3&0&1&8\\0&-5&3&4\end{pmatrix}}^{3\times 4}=\overbrace{\begin{pmatrix}9&-22&13&28\\25&2&5&56\end{pmatrix}}^{2\times 4},$$

$$\overbrace{\begin{pmatrix}-3&0&5\end{pmatrix}}^{1\times 3}\overbrace{\begin{pmatrix}2\\-4\\-3\end{pmatrix}}^{3\times 1}=-21,\qquad \overbrace{\begin{pmatrix}-3\\0\\5\end{pmatrix}}^{3\times 1}\overbrace{\begin{pmatrix}2&-4&-3\end{pmatrix}}^{1\times 3}=\overbrace{\begin{pmatrix}-6&12&9\\0&0&0\\10&-20&-15\end{pmatrix}}^{3\times 3}.$$

Correction

```
TESTS = [ (  [ [3,1,5] , [2,7,0] ] , [ [2,1,-1,0] , [3,0,1,8] , [0,-5,3,4] ] ),
          (  [ [-3,0,5 ] ]  ,  [ [2], [-4], [-3] ]  ),
          (  [ [-3],[0],[5] ] , [ [2,-4,-3] ] ) ]

for A,B in TESTS:
    print(f"A = {A}")
    print(f"B = {B}")
    n,p1 = len(A), len(A[0])
    p2,q = len(B), len(B[0])
    print(f"A est un matrice {n}x{p1}")
    print(f"B est un matrice {p2}x{q}")
    if p1==p2:
        print(f"C est un matrice {n}x{q}")
        C = [[ sum(A[i][k]*B[k][j] for k in range(p1)) for j in range(q)] for i in range(n)]
    print(f"C = {C}\n")
```

```
A = [[3, 1, 5], [2, 7, 0]]
B = [[2, 1, -1, 0], [3, 0, 1, 8], [0, -5, 3, 4]]
A est un matrice 2x3
B est un matrice 3x4
C est un matrice 2x4
C = [[9, -22, 13, 28], [25, 2, 5, 56]]

A = [[-3, 0, 5]]
B = [[2], [-4], [-3]]
A est un matrice 1x3
B est un matrice 3x1
C est un matrice 1x1
C = [[-21]]

A = [[-3], [0], [5]]
B = [[2, -4, -3]]
A est un matrice 3x1
B est un matrice 1x3
C est un matrice 3x3
C = [[-6, 12, 9], [0, 0, 0], [10, -20, -15]]
```



Exercice 5.28 (Nombres triangulaires)

La suite des nombres triangulaires est générée en additionnant les nombres naturels. Ainsi, le 7-ème nombre triangulaire est $1+2+3+4+5+6+7=28$. Les dix premiers nombres triangulaires sont les suivants : $[1,3,6,10,15,21,28,36,45,55]$

Écrire la suite des premiers 100 nombres triangulaires.

Correction

- Version naïve : $L_n=\sum_{i=1}^{n} i = 1+2+3+\cdots+n =$ `sum(range(1,n+1))`

```
L = [ sum(range(1,n+1)) for n in range(1,101)]
print(L)
```

- Mieux: $\begin{cases} L_1 = 1 \\ L_n = L_{n-1} + n \end{cases}$

```
L = [1]
for n in range(2,101):
    L.append(L[-1]+n)
print(L)
```

Vu que la division par 2 se fait uniquement sur des nombres pairs (car soit n est pair soit $n+1$ est pair), la suite est une suite d'entiers. Pour garder cette propriété il faut utiliser la division entière.

- Encore mieux: $L_n = \dfrac{n(n+1)}{2}$

```
mylist = [n*(n+1)//2 for n in range(1,101)]
print(mylist)
```

[1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171, 190, 210, 231, 253, 276, 300, 325, 351, 378, 406, 435, 465, 496, 528, 561, 595, 630, 666, 703, 741, 780, 820, 861, 903, 946, 990, 1035, 1081, 1128, 1176, 1225, 1275, 1326, 1378, 1431, 1485, 1540, 1596, 1653, 1711, 1770, 1830, 1891, 1953, 2016, 2080, 2145, 2211, 2278, 2346, 2415, 2485, 2556, 2628, 2701, 2775, 2850, 2926, 3003, 3081, 3160, 3240, 3321, 3403, 3486, 3570, 3655, 3741, 3828, 3916, 4005, 4095, 4186, 4278, 4371, 4465, 4560, 4656, 4753, 4851, 4950, 5050]

Exercice Bonus 5.29 (Nombres pentagonaux)

Un nombre pentagonal est un nombre qui peut être représenté par un pentagone https://fr.wikipedia.org/wiki/Nombre_pentagonal).

Pour tout entier $n \geq 1$, considérons la suite arithmétique $(P_n)_{n\in\mathbb{N}^*}$ de premier terme 1 et de raison 3:

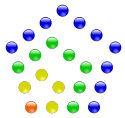
$$\begin{cases} P_1 = 1, \\ P_n = P_{n-1} + 3. \end{cases}$$

Le n-ième nombre pentagonal est la somme des n premiers termes de cette suite:

$$L_n = \sum_{i=1}^{n} P_i = 1 + 4 + 7 + \cdots + (3n-2).$$

Les dix premiers nombres pentagonaux sont les suivants: $[1, 5, 12, 22, 35, 51, 70, 92, 117, 145]$.

Écrire la suite des premiers 100 nombres pentagonaux.



Correction

```
P = [1]
for n in range(2,101):
    P.append(P[-1]+3)

L = [sum(P[:n]) for n in range(1,len(P))]
print(L)
```

On peut expliciter la suite définie par récurrence en écrivant P_n directement en fonction n: $P_n = 3n-2$ ainsi

$$L_n = \sum_{i=1}^{n} P_i = \sum_{i=1}^{n} 3i - 2 = 3\sum_{i=1}^{n} i - 2n = 3\frac{n(n+1)}{2} - 2n = \frac{n(3n-1)}{2}.$$

```
mylist = [ n*(3*n-1)//2 for n in range(1,101)]
print(mylist)
```

[1, 5, 12, 22, 35, 51, 70, 92, 117, 145, 176, 210, 247, 287, 330, 376, 425, 477, 532, 590, 651, 715, 782, 852, 925, 1001, 1080, 1162, 1247, 1335, 1426, 1520, 1617, 1717, 1820, 1926, 2035, 2147, 2262, 2380, 2501, 2625, 2752, 2882, 3015, 3151, 3290, 3432, 3577, 3725, 3876, 4030, 4187, 4347, 4510, 4676, 4845, 5017, 5192, 5370, 5551, 5735, 5922, 6112, 6305, 6501, 6700, 6902, 7107, 7315, 7526, 7740, 7957, 8177, 8400, 8626, 8855, 9087, 9322, 9560, 9801, 10045, 10292, 10542, 10795, 11051, 11310, 11572, 11837, 12105, 12376, 12650, 12927, 13207, 13490, 13776, 14065, 14357, 14652, 14950]

Exercice Bonus 5.30 (Défi Turing n°45 – Nombre triangulaire, pentagonal et hexagonal)
Les nombres triangulaires, pentagonaux et hexagonaux sont générés par les formules suivantes :

Nom	n-ème terme de la suite	Suite
triangulaire	$T_n = \frac{n(n+1)}{2}$	$1,3,6,10,15,\ldots$
pentagonal	$P_n = \frac{n(3n-1)}{2}$	$1,5,12,22,35,\ldots$
hexagonal	$H_n = n(2n-1)$	$1,6,15,28,45,\ldots$

1 est un nombre triangulaire, pentagonal et hexagonal car $1 = T_1 = P_1 = H_1$. Le suivant est 40755 car $40755 = T_{285} = P_{165} = H_{143}$. Trouver le suivant.

Correction
Un code naïf est le suivant mais on se rend compte que la complexité est trop élevée (*e.g.* l'exécution prend trop de temps) :

```
N = 10**5
T = [n*(n+1)//2 for n in range(1,N)]
P = [n*(3*n-1)//2 for n in range(1,N)]
H = [n*(2*n-1) for n in range(1,N)]
for x in T:
    if x in P and x in H:
        print(f"{x}=T_{T.index(x)}=P_{P.index(x)}=H_{H.index(x)}")
```

On va alors réfléchir différemment : pour tout $x \in T$, on cherche s'il existe $n \in \mathbb{N}$ tel que $n(3n-1) = 2x$:

$$n = \frac{1+\sqrt{1+24x}}{6} \in \mathbb{N} \iff \texttt{(1+(1+24*x)**0.5)\%6==0}$$

Si c'est le cas, on cherche s'il existe $m \in \mathbb{N}$ tel que $m(2m-1) = x$:

$$m = \frac{1+\sqrt{1+8x}}{4} \in \mathbb{N} \iff \texttt{(1+(1+8*x)**0.5)\%4==0}$$

```
cond = False
n = 0

while cond==False:
    n += 1
    x = n*(n+1)//2
    p1 = (1+(1+24*x)**0.5)
    if p1%6==0:
        h1 = (1+(1+8*x)**0.5)
        if h1%4==0:
            print(f"{x} = T_{n} = P_{int(p1/6)} = H_{int(h1/4)}")
            if x>40755 :
                cond = True

1 = T_1 = P_1 = H_1
40755 = T_285 = P_165 = H_143
1533776805 = T_55385 = P_31977 = H_27693
```

Exercice 5.31 (Table de multiplication)
Afficher la table de multiplication par $1,\ldots,10$ suivante (l'élément en position (i,j) est égal au produit ij lorsque les

indices commencent à 1) :

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Correction

```
n = 10
M = []
for i in range(1,n):
    M.append([(j+1)*(i+1) for j in range(n)])
print(M)
```

soit encore

```
n = 10
M = [ [(j+1)*(i+1) for j in range(n)] for i in range(n) ]
print(M)
```

[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [2, 4, 6, 8, 10, 12, 14, 16, 18, 20], [3, 6, 9, 12, 15, 18, 21, 24, 27, 30], [4, 8, 12, 16, 20, 24, 28, 32, 36, 40], [5, 10, 15, 20, 25, 30, 35, 40, 45, 50], [6, 12, 18, 24, 30, 36, 42, 48, 54, 60], [7, 14, 21, 28, 35, 42, 49, 56, 63, 70], [8, 16, 24, 32, 40, 48, 56, 64, 72, 80], [9, 18, 27, 36, 45, 54, 63, 72, 81, 90], [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]]

Affichage amélioré :

```
for row in M:
    for num in row:
        print(f"{num:3d}", end=" ")
    print("")
```

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

Exercice 5.32 (Défi Turing n°1 – somme de multiples)

Si on liste tous les entiers naturels inférieurs à 20 qui sont multiples de 5 ou de 7, on obtient 5, 7, 10, 14 et 15. La somme de ces nombres est 51.

Trouver la somme de tous les multiples de 5 ou de 7 inférieurs à 2013.

Correction

Reformulons le problème : on cherche à calculer la somme de tous les nombres inférieurs à 2013 divisibles par 5 ou 7. Un programme brute force (programme qui teste toutes les valeurs possibles) fonctionne très bien. Pour savoir si un nombre a est divisible par un nombre b, il faut que le reste de la division euclidienne de ces deux nombres soit égal à 0.

```
somma = 0
for i in range(1,2014):
    if (i%5)==0 or (i%7)==0:
        somma += i
print(somma)
```

ou, en générant d'abord la liste,

```
somma = sum([i for i in range(1,2014) if (i%5)==0 or (i%7)==0])
print(somma)
```

Dans tous les cas le résultat est

```
636456
```

Le problème peut être entièrement résolu en utilisant des mathématiques. Trouver la somme de tous les nombres inférieurs à 2013 divisibles par 5 ou 7, revient à faire la somme de tous les multiples de 5 inférieurs à 2013, d'y ajouter la somme de tous les multiples de 7 inférieurs à 2013 et d'y soustraire tous les multiples de 35 inférieurs à 2013 (car les deux nombres sont premiers).

```
sept = [i for i in range(0,2014,7)]
cinq = [i for i in range(0,2014,5)]
doublons = [i for i in range(0,2014,5*7)]
somma = sum(sept)+sum(cinq)-sum(doublons)
print(somma)
```

On obtient

$$\begin{aligned}\sum_{k=1}^{\lfloor\frac{2013-1}{5}\rfloor} 5k + \sum_{k=1}^{\lfloor\frac{2013-1}{7}\rfloor} 7k - \sum_{k=1}^{\lfloor\frac{2013-1}{35}\rfloor} 35k &= 5\sum_{k=1}^{402} k + 7\sum_{k=1}^{287} k - 35\sum_{k=1}^{57} k \\ &= 5\frac{402\times 403}{2} + 7\frac{287\times 288}{2} - 35\frac{57\times 58}{2} = 405015 + 289296 - 57855 = 636456.\end{aligned}$$

Exercice 5.33 (Nombres de Armstrong)

En théorie des nombres, un nombre d'Armstrong (d'après Michael F. Armstrong) est un nombre qui est la somme de ses propres chiffres, chacun élevé à la puissance du nombre de chiffres. Par exemple 153 est un nombre de Armstrong puisque $153 = 1^3 + 5^3 + 3^3$. On peut montrer qu'il n'existe que 4 nombres de Armstrong ayant 3 chiffres. Écrire la liste des 4 nombres de Armstrong.

Correction

Pour résoudre ce problème, on converti chaque nombre de 3 chiffres en chaîne de caractères, on lis les chiffres un par un, on les convertit en entier, enfin on additionne leur cube et on vérifie si on trouve encore le nombre initial :

```
Armstrong = []
for n in range(100,1000):
    S = sum([int(i)**3 for i in str(n)])
    if S==n:
        Armstrong.append(n)
print(Armstrong)
```

```
[153, 370, 371, 407]
```

ou, de façon plus compacte,

```
Armstrong = [n for n in range(100,1000) if n==sum([int(i)**3 for i in str(n)])]
print(Armstrong)
```

```
[153, 370, 371, 407]
```

Exercice 5.34 (Imprimeur et nombre d'utilisations d'un chiffres)

Un imprimeur a imprimé un livre de 1234 pages. Combien de fois il a utilisé le caractère '4' ? Autrement-dit, combien de fois le chiffre '4' apparaît en écrivant les nombres de 1 à 1234 inclus ? Répondre en une ligne (revoir par exemple l'exercice 2.10).

Astuce : créer une liste `L` de 1235 éléments telle que `L[i]` est égal au nombre de fois où le chiffre '4' a été utilisé pour écrire la page `i`.

Correction

```
print(sum( [str(page).count('4') for page in range(1,1235) ] ))
```

```
344
```

La compréhension de liste `[ ... for i in ...]` génère une liste où chaque élément est le nombre de fois que le chiffre 4 apparaît dans la représentation en chaîne de caractères de chaque nombre de 1 à 1234. En effet

- `for page in range(1,1235)` génère les nombres de 1 à 1234 inclus;
- `str(page)` convertit chaque nombre en une chaîne de caractères;
- `str(page).count('4')` compte combien de fois le chiffre 4 apparaît dans cette chaîne de caractères;
- `sum(...)` additionne tous les éléments de la liste générée par la compréhension de liste.

Attention à ne pas écrire

```
print(sum( [ '4' in str(page) for page in range(1,1235) ] ))
```

```
313
```

qui indique seulement dans combien de page on utilise le chiffre 4 indépendamment de combien de fois il sera utilisé pour écrire ce numéro de page.

Exercice 5.35 (Défi Turing n°85 – Nombres composés de chiffres différents)

Il y a 32490 nombres composés de chiffres tous différents entre 1 et 100000, par exemple 4, 72, 1468, 53920, etc. Quelle est la somme de ces nombres?

Correction

```
print(sum([n for n in range(1,100001) if len(str(n))==len(set(str(n)))]))
```

```
1520464455
```

- `range(1,100001)` génère une séquence de nombres de 1 à 100000 inclus;
- `str(n)` convertit chaque nombre en une chaîne de caractères pour faciliter le traitement des chiffres;
- `set(str(n))` crée un ensemble des chiffres uniques présents dans le nombre. L'utilisation d'un ensemble garantit que seuls les chiffres uniques sont pris en compte.
- `len(str(n))` donne le nombre total de chiffres dans le nombre d'origine.
- La condition `len(str(n))==len(set(str(n)))` vérifie si le nombre de chiffres uniques est égal au nombre total de chiffres. Si c'est le cas, cela signifie qu'il n'y a pas de chiffres répétés.
- Enfin, `sum(...)` additionne tous ces nombres pour donner le total.

Exercice Bonus 5.36 (Pydéfi – Piège numérique à Pokémons)

Ossatueur et Mewtwo sont passionnés par les nombres. On le sait peu. Le premier apprécie tout particulièrement les multiples de 7: 7, 14, 21... Le second adore les nombres dont la somme des chiffres vaut exactement 11: 29, 38, 47... Pour les attirer, vous chantonnez les nombres qu'ils préfèrent. Quels sont les nombres entiers positifs inférieurs à 1000 qui plaisent à la fois à Ossatueur et Mewtwo?

Source: https://pydefis.callicode.fr/defis/PokeNombresCommuns/txt

Exercice Bonus 5.37 (Pydéfi – Nos deux chiffres préférés)

Calculer la somme des nombres compris entre deux bornes L = 140 et R = 1007 (incluses) qui contiennent le chiffre 7 ou le chiffre 4 (ou les deux).

Testez votre code: si les bornes sont L = 10 et R = 54, le total à donner est 652, car la liste des nombres à ajouter est [14, 17, 24, 27, 34, 37, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 54].

Source: https://pydefis.callicode.fr/defis/ChiffresPreferes/txt

Exercice 5.38 (`sum()` `len()` `all()` `any()`)

On donne une liste L contenant des entiers. Utilisez des listes en compréhension et les fonctions `sum()`, `len()`, `all()` et `any()` pour répondre aux questions suivantes.

1. **Compter les éléments** de L qui satisfont une propriété donnée (par exemple, être divisibles par 3).
2. Vérifier si **tous les éléments** de L satisfont une propriété donnée.
3. Vérifier si **au moins un élément** de L satisfait une propriété donnée.

Par exemple, pour L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:

- Nombre d'éléments divisibles par 3: 4.

- Tous les éléments sont divisibles par 3 ? `False`.
- Au moins un élément est divisible par 3 ? `True`.

Correction

- `sum()` : additionne les valeurs d'une liste ou d'un itérable. Elle peut être utilisée pour compter les occurrences où une condition est vraie, car `True` est équivalent à 1 et `False` à 0 en Python.
- `len()` : retourne le nombre d'éléments d'une liste ou d'un itérable.
- `all()` : retourne `True` si tous les éléments de la liste ou de l'itérable sont vrais. Si au moins un élément est `False`, retourne `False`.
- `any()` : retourne `True` si au moins un élément de la liste ou de l'itérable est vrai. Si tous les éléments sont `False`, retourne `False`.

```
>>> # Liste donnée
>>> L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> # 1. Nombre d'éléments divisibles par 3
>>> len([ell for ell in L if ell % 3 == 0])  # Filtrer
4
>>> sum([ell % 3 == 0 for ell in L])  # True est équivalent à 1
4

>>> # 2. Tous les éléments sont divisibles par 3 ?
>>> all([ell % 3 == 0 for ell in L])
False

>>> # 3. Au moins un élément est divisible par 3 ?
>>> any([ell % 3 == 0 for ell in L])
True
```

Exercice 5.39 (Sélection de nombres)

Créer une liste contenant tous les entiers compris entre 10 et 999 qui répondent aux critères suivants : l'entier se termine par le chiffre 3, le chiffre des dizaines est pair, et la somme de ses chiffres est strictement supérieure à 15.

Correction

Première méthode : on considère tous les entiers compris entre 10 et 999 et on vérifie s'ils satisfont les propriétés demandées.

- Les entiers appartiennent à l'ensemble $\{10, 11, \ldots, 999\}$, soit `range(10,1000)`.
- Pour vérifier qu'un entier n se termine par 3, on peut écrire `n%10==3` (ou encore `int(str(n)[-1])==3` ou `str(n)[-1]=='3'`).
- Pour vérifier que le chiffre des dizaines d'un entier n (qui a au moins 2 chiffres) est pair, on peut écrire `int(str(n)[-2])%2==0` ou `str(n)[-2] in "02468"` ou encore `((n//10)%10) %2==0`.
- Pour vérifier que la somme des chiffres d'un entier n est supérieure à 15, on parcourt les chiffres un par un (`for c in str(n)`) et on les converti en entier (`int(c)`), enfin on additionne les éléments de cette liste (*cf.* exercice 5.10), ce qui se traduit par `sum([int(c) for c in str(n)])>15`.

On peut ainsi écrire

```
[ n for n in range(10, 1000) if n%10==3 and ((n//10)%10) %2==0 and sum([int(c) for c in str(n)])>15 ]
```

On peut simplifier légèrement en remarquant que les entiers qui se terminent par 3 sont de la forme `range(13, 1000, 10)`. Enfin, comme le chiffre des dizaines doit être pair, les entiers concernés sont de la forme `range(23, 1000, 20)` :

```
print( [ n for n in range(23,994,20) if sum([int(c) for c in str(n)])>15 ] )
```

```
[583, 683, 763, 783, 863, 883, 943, 963, 983]
```

Deuxième méthode : on construit les entiers directement à partir de leurs chiffres

$$n = c \times 10^2 + d \times 10 + u \qquad \text{avec} \qquad u = 3, \qquad d \in \{0,2,4,6,8\}, \qquad c \in \{0,1,2,3,4,5,6,7,8,9\}.$$

On impose la condition $c + d + 3 > 15$, soit $c > 12 - d$. Notons que, comme $c < 10$, cette condition n'a de sens que si $13 - d \leq 9$, ainsi $d \in \{4,6,8\}$ et $c \in \{5,6,7,8,9\}$ avec $c > 12 - d$.

```
L = sorted([ 100*c+10*d+3 for d in range(4,9,2) for c in range(12-d+1,10)])
print(L)
```

```
[583, 683, 763, 783, 863, 883, 943, 963, 983]
```

Cet algorithme peut être exécuté “à la main” comme suit :

- pour $d = 4$, $c = 9$ donc $n = 943$;
- pour $d = 6$, $c = 7, 8, 9$ donc $n = 763, 863, 963$;
- pour $d = 8$, $c = 5, 6, 7, 8, 9$ donc $n = 583, 683, 783, 883, 983$.

Exercice Bonus 5.40 (Pydéfi – SW III : L’ordre 66 ne vaut pas 66...)

Lorsque Palpatine prend le pouvoir, il entame la destruction des Jedis en envoyant un ordre à tous les clones : l’ordre 66. Les clones se retournent alors contre les Jedis. C’est du moins ce que prétend la légende cinématographique.

La réalité est un peu différente et s’il y a bien un ordre particulier à donner aux clones, ce n’est pas l’ordre 66. Pour le rendre difficile à découvrir, les Kaminoans ont utilisé les talents de calcul mental de Jango Fett en apprenant aux clones une liste de propriétés. Si le numéro satisfait ces propriétés, alors les clones se retourneront contre les Jedis.

Voici la liste des propriétés enseignées aux clones :

- l’ordre est un nombre à 4 chiffres, tous impairs
- chaque chiffre du nombre est strictement plus petit que le suivant (par exemple 3579 convient, mais pas 3557 ou 3157)
- le produit des chiffres du nombre est un nouveau nombre qui ne contient que des chiffres impairs
- la somme des chiffres du nombre est un nouveau nombre qui ne contient que des chiffres pairs

Trouvez le seul numéro d’ordre qui convient et provoque l’attaque des clones contre les Jedis.

Source : https://pydefis.callicode.fr/defis/Ordre66/txt

Exercice Bonus 5.41 (Pydéfi – Constante de Champernowne)

La constante de Champernowne est un nombre compris entre 0 et 1, dont le dévelopopement décimal est obtenu en écrivant successivement les nombre entiers. Elle commence ainsi :

$$0,123456789101112131415151718192021 22\ldots$$

Numérotons les chiffres situés après la virgule. Le chiffre 1 est 1, le chiffre 9 est 9, le chiffre 10 est 1 et le chiffre 11 est 0 etc.

Donner la somme des chiffres de $n_1 = 104$ à $n_2 = 156$ inclus. Par exemple, si $n_1 = 11$ et $n_2 = 21$, la réponse à donner serait alors $0+1+1+1+2+1+3+1+4+1+5 = 20$.

Source : https://pydefis.callicode.fr/defis/Champernowne/txt

Exercice Bonus 5.42 (Défi Turing n° 40 – La constante de Champernowne)

La constante de Champernowne est un nombre irrationnel créé en concaténant les entiers positifs :

$$0,12345678910111213141515171819202 1\ldots$$

On peut voir que le 12-ème chiffre de la partie fractionnaire est 1.

Si d_n représente le n-ième chiffre de la partie fractionnaire, quelle est la valeur de l’expression suivante ?

$$d_1 \times d_{10} \times d_{100} \times d_{1000} \times d_{10000} \times d_{100000} \times d_{1000000} \times d_{10000000} \times d_{100000000}$$

Correction

```
indices = [1, 10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000]
product = 1
length = 0
digits = 1

for idx in indices:
    while True:
        count = 9 * 10**(digits - 1) * digits
        if idx <= length + count:
            number = 10**(digits - 1) + (idx - length - 1) // digits
```

```
            digit = int(str(number)[(idx - length - 1) % digits])
            product *= digit
            break
        length += count
        digits += 1

print(product)
```

```
11760
```

Exercice Bonus 5.43 (Pydéfi – Série décimée...)

Considérons la série harmonique

$$S(n) = \sum_{i=1}^{i=n} \frac{1}{i}$$

Soit $T(n)$ la série obtenue en excluant de $S(n)$ toutes les fractions qui contiennent le chiffre 9 au dénominateur (par exemple 1/9 et 1/396). Elle s'appelle "série de Kempner". Que vaut $T(242)$?

Source : https://pydefis.callicode.fr/defis/SuiteDecimation/txt

L'intérêt de cette suite réside dans le fait que contrairement à la série harmonique, elle converge. Ce résultat fut démontré en 1914 par Aubrey J. Kempner : le nombre d'entiers à n chiffres, dont le premier est compris entre 1 et 8 et les $n-1$ suivants entre 0 et 8, est $8 \times 9^{n-1}$, et chacun d'eux est minoré par 10^{n-1}, donc la série est majorée par la série géométrique $8\sum_{n=1}^{\infty}\left(\frac{9}{10}\right)^{n-1} = 80$.

Exercice Bonus 5.44 (Pydéfi – Désamorçage de bombe à distance (II))

Après leur cuisant échec face à Black Widow, les Maîtres du Mal ont placé une nouvelle bombe dévastatrice à Los Angeles. Cette fois-ci, impossible de s'en approcher : Œil de faucon doit la désamorcer à distance, en coupant deux fils avec son arc et ses flèches. Pour connaître les fils à couper, il y a un certain nombre d'instructions à suivre.

Les fils de la bombe sont numérotés. Supposons pour l'exemple qu'il n'y ait «que» 500 fils numérotés de 1 à 500. Pour connaître les deux fils à couper, on procède par élimination :

- Conserver les fils dont le numéro est multiple de 5 ou de 7. Les fils conservés sont :
  `5, 7, 10, 14, 15, 20, 21, 25, 28, 30, 35, 40, 42, 45, 49, 50, 55, 56, 60, 63, 65, 70, 75, 77, 80, 84, 85, 90, 91, 95, 98, 100, 105, 110, 112, 115, 119, 120, 125, 126, 130, 133, 135, 140, 145, 147, 150, 154, 155, 160, 161, 165, 168, 170, 175, 180, 182, 185, 189, 190, 195, 196, 200, 203, 205, 210, 215, 217, 220, 224, 225, 230, 231, 235, 238, 240, 245, 250, 252, 255, 259, 260, 265, 266, 270, 273, 275, 280, 285, 287, 290, 294, 295, 300, 301, 305, 308, 310, 315, 320, 322, 325, 329, 330, 335, 336, 340, 343, 345, 350, 355, 357, 360, 364, 365, 370, 371, 375, 378, 380, 385, 390, 392, 395, 399, 400, 405, 406, 410, 413, 415, 420, 425, 427, 430, 434, 435, 440, 441, 445, 448, 450, 455, 460, 462, 465, 469, 470, 475, 476, 480, 483, 485, 490, 495, 497, 500`
- Dans ce qui reste, conserver les fils dont le chiffre des dizaines est inférieur ou égal au chiffre des unités. Il reste les fils :
  `5, 7, 14, 15, 25, 28, 35, 45, 49, 55, 56, 77, 100, 105, 112, 115, 119, 125, 126, 133, 135, 145, 147, 155, 168, 189, 200, 203, 205, 215, 217, 224, 225, 235, 238, 245, 255, 259, 266, 300, 301, 305, 308, 315, 322, 325, 329, 335, 336, 345, 355, 357, 378, 399, 400, 405, 406, 413, 415, 425, 427, 434, 435, 445, 448, 455, 469, 500`
- Dans ce qui reste, conserver les fils dont le voisin de droite a un chiffre des unités strictement plus petit que 5 (il faudra opérer en parcourant les fils de gauche à droite). Le fil le plus à droite n'est pas conservé. Après cette opération, il restera les fils :
  `7, 77, 105, 126, 189, 200, 217, 266, 300, 315, 399, 406, 427, 469`
- Dans ce qui reste, conserver les fils dont le chiffre des dizaines est impair. Il reste :
  `77, 217, 315, 399`
- Une fois ces opérations faites, tous les fils ont été écartés sauf un nombre pair d'entre eux. Pour désamorcer la bombe, il faut couper les deux fils du milieu dans ceux qui restent. Dans notre cas, il ne reste que quatre fils, il faut donc couper les fils 217 et 315.

En réalité, **la bombe contient 4200 fils**. Pour résoudre le défi, indiquez à Œil de faucon les numéros des deux fils à couper.

Source : https://pydefis.callicode.fr/defis/Desamorcage02/txt

Exercice Bonus 5.45 (Défi Turing n°29 – puissances distincts)

Considérons a^b pour $2 \leq a \leq 5$ et $2 \leq b \leq 5$:

$$\begin{array}{llll} 2^2 = 4, & 2^3 = 8, & 2^4 = 16, & 2^5 = 32 \\ 3^2 = 9, & 3^3 = 27, & 3^4 = 81, & 3^5 = 243 \\ 4^2 = 16, & 4^3 = 64, & 4^4 = 256, & 4^5 = 1024 \\ 5^2 = 25, & 5^3 = 125, & 5^4 = 625, & 5^5 = 3125 \end{array}$$

Si l'on trie ces nombres dans l'ordre croissant, en supprimant les répétitions, on obtient une suite de 15 termes distincts : $[4, 8, 9, 16, 25, 27, 32, 64, 81, 125, 243, 256, 625, 1024, 3125]$.

Combien y a-t-il de termes distincts dans la suite obtenue comme ci-dessus pour $2 \leq a \leq 1000$ et $2 \leq b \leq 1000$?

Correction

```
>>> print (len(set(a**b for a in range(2,1001) for b in range(2,1001))))
977358
```

Exercice Bonus 5.46 (Défi Turing n°94 – Problème d'Euler n° 92)

Une suite d'entiers est créée de la façon suivante : le nombre suivant de la liste est obtenu en additionnant les carrés des chiffres du nombre précédent :

$$44 \xrightarrow[4^2+4^2]{} 32 \xrightarrow[3^2+2^2]{} 13 \xrightarrow[1^2+3^2]{} 10 \xrightarrow[1^2+0^2]{} 1 \xrightarrow[1^2]{} 1$$

$$85 \xrightarrow[8^2+5^2]{} 89 \xrightarrow[8^2+9^2]{} 145 \xrightarrow[1^2+4^2+5^2]{} 42 \xrightarrow[4^2+2^2]{} 20 \xrightarrow[2^2+0^2]{} 4 \xrightarrow[4^2]{} 16 \xrightarrow[1^2+6^2]{} 37 \xrightarrow[3^2+7^2]{} 58 \xrightarrow[5^2+8^2]{} 89$$

On peut voir qu'une suite qui arrive à 1 ou 89 restera coincée dans une boucle infinie. Le plus incroyable est qu'avec n'importe quel nombre de départ strictement positif, toute suite arrivera finalement à 1 ou 89.

Combien de nombres de départ inférieurs ou égal à 5 millions arriveront à 89 ?

Correction

```
d = {1:0,89:0}
for i in range(1,5000000+1):
    while i!=1 and i!=89:
        i = sum([int(i)**2 for i in str(i)])
    d[i] += 1
print(d)
```

```
{1: 704156, 89: 4295844}
```

Exercice Bonus 5.47 (Pydéfi – Le pistolet de Nick Fury)

Recherche du fonctionnement périodique : le pistolet de Nick Fury émet des impulsions successives dont l'intensité varie selon une loi mathématique. Pour calculer l'intensité de l'impulsion suivante, il suffit d'écrire en binaire l'intensité de l'impulsion émise, de renverser l'écriture de ce nombre binaire (lire de droite à gauche), de convertir le nombre obtenu en base 10 puis de lui ajouter 2 et recommencer.

Sur le pistolet, on peut régler l'intensité de l'impulsion initiale. Par exemple, si le pistolet est réglé sur 39, alors, lors d'un tir, les impulsions émises auront pour intensité

$$39 \xrightarrow[\text{binaire}]{} 100111 \xrightarrow[\text{miroir}]{} 111001 \xrightarrow[\text{base 10}]{} 57 \xrightarrow[+2]{}$$

$$59 \xrightarrow[\text{binaire}]{} 111011 \xrightarrow[\text{miroir}]{} 110111 \xrightarrow[\text{base 10}]{} 55 \xrightarrow[+2]{}$$

$$57 \xrightarrow[\text{binaire}]{} 111001 \xrightarrow[\text{miroir}]{} 100111 \xrightarrow[\text{base 10}]{} 39 \xrightarrow[+2]{}$$

$$41 \xrightarrow[\text{binaire}]{} 101001 \xrightarrow[\text{miroir}]{} 100101 \xrightarrow[\text{base 10}]{} 37 \xrightarrow[+2]{} 39$$

On constate que pour le réglage 39, les amplitudes sont périodiques et la période est 4 ($39 \to 59 \to 57 \to 39$).

Voici un autre exemple, obtenu avec une impulsion initiale de 86 : $86 \to 55 \to 61 \to 49 \to 37 \to 43 \to 55$. Bien qu'on ne retourne jamais à la valeur 86, on obtient aussi un cycle, de longueur 5.

En revanche, pour certaines valeurs, l'amplitude n'est jamais périodique, et le comportement du pistolet est imprévisible. Si Nick le règle sur une telle valeur, le pistolet peut exploser dès qu'il a changé plus de 1024 fois d'intensité.

Afin d'améliorer l'arme de Nick Fury et ainsi rendre service au Shield, il convient de ne permettre que les réglages des valeurs de départ qui donnent lieu à un comportement périodique.

Défi : donner la séquence de toutes les valeurs convenables comprises entre 1 et 500 c'est-à-dire celles qui donnent lieu à un comportement périodique.

Source : https://pydefis.callicode.fr/defis/PistoletFury

Exercice Bonus 5.48 (Pydéfi – Les nombres heureux)

Les nombres sont parfois pourvus de qualificatifs surprenants. Il existe en effet des nombres heureux et des nombre malheureux. Pour savoir si un nombre est heureux, il faut calculer la somme des carrés de ses chiffres, et recommencer avec le résultat. Si on finit par tomber sur 1, alors le nombre est heureux. Sinon, il est malheureux.

Par exemple, le nombre 109 est heureux. En effet :

$$109 \to 1^2+0^2+9^2 = 82 \to 8^2+2^2 = 68 \to 6^2+8^2 = 100 \to 1^2+0^2+0^2 = \boxed{1}$$

Par contre, 106 est malheureux. En effet :

$$\begin{aligned}106 &\to 1^2+0^2+6^2 = \boxed{37} \to 3^2+7^2 = 58 \to 5^2+8^2 = 89 \to 8^2+9^2 = 145 \to 1^2+4^2+5^2 = 42 \to 4^2+2^2 = 20\\ &\to 2^2+0^2 = 4 \to 4^2 = 16 \to 1^2+6^2 = \boxed{37}\end{aligned}$$

Le nombre 37 a déjà été obtenu en début de séquence, on sait donc que la série $37, 58, 89, 145, 42, 20, 4, 16$ va se répéter indéfiniment. Le nombre 1 ne sera donc jamais atteint.

Défi : l'entrée du problème est la donnée de deux bornes mini et maxi. Il faut répondre en donnant la liste des nombres heureux compris entre ces deux bornes (incluses), par ordre croissant.

Testez votre code : par exemple, si les bornes données étaient mini=109 et maxi=141, il faudrait répondre en indiquant $(109, 129, 130, 133, 139)$.

Source : https://pydefis.callicode.fr/defis/NombresHeureux

Exercice Bonus 5.49 (Pydéfi – Le problème des boîtes à sucres)

La société Syntactic Sugar emballe depuis des années des sucres cubiques en boîtes parallélépipédiques. La tradition veut que chaque boîte contienne 252 sucres disposés en 4 couches de 7×9 sucres. Les sucres étant cubiques, et les boîtes de la société Syntactic Sugar pouvant s'ouvrir sur n'importe quelle face, on peut tout aussi bien considérer qu'il s'agit d'une boîte contenant 7 couches de 4×9 sucres ou encore 9 couches de 7×4 sucres.

Un beau matin, mu par un irrépressible besoin d'innovation, le service commercial de la société Syntactic Sugar décide que des boîtes contenant 3 couches de 7×12 sucres (et donc toujours 252 sucres) seraient bien plus attractives. Il s'en suivit de nombreuses querelles sur la forme des boîtes, certains souhaitant plutôt fabriquer maintenant des boîtes contenant 3 couches de 14×6 sucres...

Jusqu'au moment où l'idée ne put que germer : et si on changeait le nombre de sucres par boîte ?

Une boîte raisonnable contenant entre 137 et 479 sucres, quel nombre de sucres choisir pour qu'il n'y ait qu'une seule forme de boîte possible, et par là même couper court aux tergiversations du service commercial, sachant que sur toutes les dimensions, on met au minimum 2 sucres (c'est à dire qu'une boîte de dimensions 15x15x1 contenant 225 sucres ne conviendrait pas, car aurait une dimension égale à 1).

Par exemple, en faisant des boîtes de 385 sucres, on est dans l'obligation de réaliser de boîtes de dimensions $5 \times 7 \times 11$. Aucune autre forme de boîte ne convient.

Défi : donner la liste des nombres de sucres qui imposent une taille de boîte unique. Notez que 316, qui impose une taille de boîte de $2 \times 2 \times 79$ est convenable, quoique fort peu pratique.

Source : https://pydefis.callicode.fr/defis/BoitesSucres

Exercice Bonus 5.50 (Anniversaires)

Un dictionnaire `naissances` associe des prénoms (clés) à des mois de naissance (valeurs entières : 1 pour janvier, 2 pour février, etc.). Exemple : `naissances = {'Nicolas': 10, 'Antoine': 7, 'Camille': 11}`. Écrire une liste en compréhension qui contient les prénoms des personnes nées au mois donné `mois`.

Source : https://codex.forge.apps.education.fr/exercices/anniversaires/

Correction

```
naissances = {"Nicolas": 10, "Antoine": 7, "Camille": 11}
mois = 7
resultat = [prenom for prenom, m in naissances.items() if m == mois]
print(resultat)
```

```
['Antoine']
```

Exercice Bonus 5.51 (Triplets pytagoriciens)

Le triplet d'entiers naturels non nuls (a, b, c) est pytagoricien si $a^2 + b^2 = c^2$. Pour $c = 2020$ il existe 4 triplets pytagoricien différents : $(400, 1980)$, $(868, 1824)$, $(1212, 1616)$ et $(1344, 1508)$

Pour chaque c in $[2010, 2020]$ calculer combien de triplets pytagoriciens existent.

Correction

```
dico = { c : [(a,b) for a in range(1,c) for b in range(a,c) if a**2+b**2==c**2] for c in
↪  range(2010,2021)}
print(dico) # Pour vérification
dico1 = { c : len(v) for (c,v) in dico.items() }
print(dico1)
```

```
{2010: [(1206, 1608)], 2011: [], 2012: [], 2013: [(363, 1980)], 2014: [(1064, 1710)], 2015: [(496,
↪  1953), (775, 1860), (1023, 1736), (1209, 1612)], 2016: [], 2017: [(792, 1855)], 2018: [(1118,
↪  1680)], 2019: [(1155, 1656)], 2020: [(400, 1980), (868, 1824), (1212, 1616), (1344, 1508)]}
{2010: 1, 2011: 0, 2012: 0, 2013: 1, 2014: 1, 2015: 4, 2016: 0, 2017: 1, 2018: 1, 2019: 1, 2020: 4}
```

Exercice Bonus 5.52 (Dictionnaire ordonné)

Soit `L` une liste d'entiers positifs. Construite la liste `Y` des valeurs uniques de `L` triées par leur fréquence. Si deux valeurs apparaissent le même nombre de fois, les trier du plus petit au plus grand.

Par exemple, si L=[1, 2, 2, 2, 3, 3, 7, 7, 9] alors Y=[1, 9, 3, 7, 2].

Correction

```
L = [1, 2, 2, 2, 3, 3, 7, 7, 9]
# dictionnaire { valeur : fréquence }
dico = { i : L.count(i) for i in set(L) }
print(dico)
# sorted() sur (fréquence, valeur) pour départager les ex-aequo par valeur croissante
dico_sorted = { k:v for k,v in sorted(dico.items(), key=lambda item: (item[1], item[0]))}
print(dico_sorted)
# on extrait juste les clés
Y = [k for k in dico_sorted.keys()]
print(Y)
```

```
{1: 1, 2: 3, 3: 2, 7: 2, 9: 1}
{1: 1, 9: 1, 3: 2, 7: 2, 2: 3}
[1, 9, 3, 7, 2]
```

CHAPITRE 6

Fonctions

Une fonction est un ensemble d'instructions regroupées sous un nom et s'exécutant à la demande. Elle peut recevoir des paramètres et peut renvoyer des objets. Les fonctions sont essentielles pour diviser le code en sections plus petites, ce qui facilite la compréhension de chaque partie et permet la réutilisation du même code.

En respectant deux règles fondamentales lors de l'écriture d'une fonction, vous améliorerez la qualité de votre code :

1. **DRY** : ne vous répétez pas (*Don't Repeat Yourself* en anglais). Si vous vous retrouvez à copier et coller souvent des morceaux de code, cela indique qu'il est possible d'isoler ces lignes de code pour les réutiliser plus efficacement.
2. **Responsabilité Unique** : chaque fonction doit avoir un seul objectif pour une meilleure lisibilité. En séparant les responsabilités, vous pouvez attribuer un nom plus descriptif à la fonction, déboguer le code plus facilement et améliorer la compréhension globale du code. Un code bien écrit doit être suffisamment lisible pour que d'autres le comprennent sans avoir besoin de trop de commentaires.

En suivant ces principes, vous obtiendrez un code mieux structuré, plus lisible et plus facile à maintenir.

6.1. Fonctions prédéfinies

De nombreuses fonctions sont déjà disponibles dans Python. Nous en avons déjà utilisée un petit nombre :

- fonctions de bases comme `abs()` `divmod()` `len()` `max()` `min()` `sum()` `round()`,
- conversions de types de variables comme `int()` `float()` `str()` `type()`,
- conversions de structures de données comme `list()` `set()` `tuple()`.

La liste pour la dernière version de Python est disponible ici : https://docs.python.org/3/library/functions.html

```
# A
abs()
aiter()
all()
any()
anext()
ascii()

# B
bin()
bool()
breakpoint()
bytearray()
bytes()

# C
callable()
chr()
classmethod()
compile()
complex()

# D
delattr()
dict()
dir()
divmod()

# E
enumerate()
eval()
exec()

# F
filter()
float()
format()
frozenset()

# G
getattr()
globals()

# H
hasattr()
hash()
help()
hex()

# I
id()
input()
int()
isinstance()
issubclass()
iter()

# L
len()
list()
locals()

# M
map()
max()
memoryview()
min()

# N
next()

# O
object()
oct()
open()
ord()

# P
pow()
print()
property()

# R
range()
repr()
reversed()
round()

# S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

# T
tuple()
type()

# V
vars()

# Z
zip()

# _
__import__()
```

6.1.1. Utilisation d'une fonction

Pour utiliser une fonction il suffit d'écrire le nom de la fonction suivi d'autant de valeurs nécessaires entre parenthèse. Par exemple la fonction "puissance" :

```
p = pow(2, 3) # 2^3 = 8
print(p)
```

```
8
```

Help : il arrive régulièrement de se rappeler du nom d'une fonction, mais pas forcément de ce qu'elle fait ou pas forcément de ces arguments. La fonction `help` est là pour ça. Si on exécute `help(nomFonction)`, cela affichera la documentation de cette fonction résumant : son but, des recommandations d'utilisation, la liste et la description des paramètres, parfois des exemples. Voici un exemple avec la fonction `pow` :

```
>>> help(pow)
Help on built-in function pow in module builtins:

pow(base, exp, mod=None)
    Equivalent to base**exp with 2 arguments or base**exp % mod with 3 arguments

    Some types, such as ints, are able to use a more efficient algorithm when
    invoked using the three argument form.
```

On voit que deux paramètres sont nécessaires (ici la base et l'exposant), mais il est possible de lui en passer 3 (pour calculer le reste modulo le nouveau paramètre) :

```
p = pow(2, 3)   # 2^3 = 8
q = pow(2, 3, 5) # 2^3 mod 5 = 3
print(p, q)
```

```
8 3
```

Appelle selon l'ordre de définition : lors de la transmission des valeurs à une fonction (dans l'exemple les nombres 2,3), celle-ci associe la première valeur au premier paramètre, et ainsi de suite, dans l'ordre où elles sont placées entre les parenthèses. Ces paramètres sont appelés des **paramètres positionnels**, et nous devons fournir les valeurs dans le même ordre que les paramètres de la fonction.

```
p = pow(2, 3) # 2^3 = 8
q = pow(3, 2) # 3^2 = 9
print(p,q)
```

```
8 9
```

Les paramètres nommés : lors de l'appel d'une fonction, il est possible de **préciser aussi le nom avec la valeur du paramètre**. Cela peut s'avérer utile pour une fonction qui accepte plusieurs paramètres. En effet, il n'est pas toujours facile de se souvenir de l'ordre exact des paramètres ni de leur signification. En nommant les paramètres au moment de l'appel, leur rôle devient plus explicite et nous ne sommes plus obligés de respecter l'ordre de leur déclaration

```
# 2^3 mod 5 = 3
# pow(2, 3, 5)
p = pow(base=2, exp=3, mod=5)
q = pow(exp=3, mod=5, base=2)
print(p, q)
```

```
3 3
```

Vous pouvez appeler un fonction en passant des paramètres (sans spécifier le nom) et des paramètres nommés. Dans ce cas, les paramètres nommés doivent être placés à la fin de la liste des paramètres.

```
p = pow(2, mod=5, exp=3) # 2^3 mod 5 = 3
print(p)
```

```
3
```

Aucun argument ne peut recevoir de valeur plus d'une fois. Faites donc bien attention à ne pas passer une valeur à un argument sans le nommer puis à repasser cette valeur en le nommant par inattention.

Valeurs par défaut : les fonctions peuvent avoir une valeur par défaut pour un ou plusieurs de ses paramètres. Ainsi, lorsqu'on appelle la fonction, on peut omettre de donner une valeur pour ces paramètres. Cela permet de rendre certains paramètres optionnels. Bien-sûr, si une valeur est explicitement passée, elle prendra le pas sur la valeur par défaut définie. C'est le cas du paramètre `mod` qui, par défaut, est `None` :

```
# par default mod=None donc on calcule simplement 2^3 = 8
pPos = pow(2, 3)
pNom = pow(base=2, exp=3)

# on écrase la valeur par default de mod et on calcule 2^3 mod 5 qui est égal à 3
qPos = pow(2, 3, 5)
qNom = pow(exp=3, mod=5, base=2)

print(pPos, pNom, qPos, qNom)
```

```
8 8 3 3
```

Tous les paramètres n'ont pas forcement des valeurs par défaut (par exemple les paramètres `base` et `exp`).

EXEMPLE (LA FONCTION PRÉDÉFINIE PRINT())
Nous avons déjà utilisé des paramètres par défaut (sans le savoir) ainsi que des paramètres optionnels, par exemple avec la fonction `print()`.

Généralement nous utilisons la fonction `print()` comme suit:

```
print("Python est puissant")
```

```
Python est puissant
```

Ici, la fonction `print()` affiche la chaîne de caractères contenue dans les guillemets simples. Elle prend donc un seul paramètre.

Pour connaître tous les paramètres de cette fonction utilisons la fonction `help`:

```
>>> help(print)
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
```

On voit que la fonction `print()` accepte 5 paramètres:

- `value` - valeur(s) à imprimer;
- `sep` (optionnel) - séparateur entre les objets (s'ils sont plusieurs) lors de l'affichage, par défaut c'est une espace;
- `end` (optionnel) - ce qui est ajouté à la fin, par défaut c'est une nouvelle ligne (`\n`);
- `file` (optionnel) - l'endroit où les valeurs sont imprimées. La valeur par défaut est `sys.stdout` (écran);
- `flush` (optionnel) - booléen spécifiant si la sortie doit être mise en mémoire tampon ou en mémoire vive. La valeur par défaut est `False`.

Dans l'exemple ci-dessous, l'instruction `print()` n'inclut que l'objet à imprimer. Ici, la valeur de `end` n'est pas utilisée. Elle prend donc la valeur par défaut (`\n`): Nous obtenons donc la sortie sur deux lignes différentes.

```
print("Bonjour !")
print("Il pleut aujourd'hui")
```

```
Bonjour !
Il pleut aujourd'hui
```

Maintenant nous ajoutons le `end= ' '` après la fin de la première instruction `print()`. Ainsi, nous obtenons la sortie sur une seule ligne séparée par un espace.

```
print("Bonjour !", end= ' ')
print("Il pleut aujourd'hui")
```

```
Bonjour ! Il pleut aujourd'hui
```

Dans l'exemple ci-dessous, l'instruction `print()` comprend plusieurs éléments séparés par une virgule. Vous remarquerez que nous avons utilisé le paramètre facultatif `sep= ". "` à l'intérieur de l'instruction `print()`. Par conséquent, la sortie comprend des éléments séparés par des points et non par des virgules:

```
print('New Year', 2023, 'See you soon!', sep= '. ')
```

```
New Year. 2023. See you soon!
```

6.2. Définition et utilisation d'une fonction personnelle

On peut définir nos propres fonctions au moyen de la commande `def` et les utiliser dans plusieurs scripts. Une fonction possèdes trois parties: un nom `FunctionName`, des paramètres (d'entrée `parameters` et/ou de sortie `values`) et un corps `statements`. La syntaxe est la suivante:

Déclaration avec `def` et utilisation.

Declaration

```
def FunctionName(parameters):
    ...
    return values
```

Utilisation

```
resultat = FunctionName(arguments)
# ou
resultat = FunctionName(parameters=arguments)
```

- La déclaration d'une nouvelle fonction commence par le mot-clé `def`.
- Ensuite, toujours sur la même ligne, vient le nom de la fonction (ici `FunctionName`) suivi des paramètres formels de la fonction `parameters`, placés entre parenthèses, le tout terminé par deux-points (on peut mettre autant de paramètres formels qu'on le souhaite et éventuellement aucun).
- Une fois la première ligne saisie, on appuie sur la touche «Entrée»: le curseur passe à la ligne suivante avec une indentation. On écrit ensuite les instructions.
- Dès que Python atteint l'instruction `return something`, il renvoie l'objet `something` et abandonne aussitôt après l'exécution de la fonction (on parle de code mort pour désigner les lignes qui suivent l'instruction `return`). Cela peut être très pratique par exemple dans une boucle `for`: dès qu'on a le résultat voulu, on le renvoie sans avoir besoin de finir la boucle. Si l'instruction `return` est absente, la fonction renvoie l'objet `None` (ce sera le cas lorsqu'on passe un objet mutable et la fonction le modifie directement sans renvoyer un nouvel objet, par exemple une liste).

La partie `FunctionName(parameters)` s'appelle la signature de la fonction.

EXEMPLE

Faisons un parallèle avec les fonctions que vous avez l'habitude d'utiliser en mathématiques: on défini d'abord une fonction $f : \mathbb{R} \to \mathbb{R}$, puis on l'évalue pour une valeur de x donnée, par exemple $x = 2$:

Mathématiques

- Déclaration $f : x \mapsto 2x^7 - x^6 + 5x^5 - x^4 + 9x^3 + 7x^2 + 8x - 1$
- Utilisation $y = f(2)$ (on affecte la valeur 451 à y).

Python

- Déclaration

```
def f(x):
    return 2*x+1
```

- Utilisation

```
y = f(2) # avec print(y) on trouve bien 451
# ou bien
y = f(x=2)
```

Faisons encore un parallèle avec une fonction mathématique de plusieurs variables cette fois-ci:

Mathématiques

- Déclaration $f : (x, y) \mapsto x^2 - y$
- Utilisation $z = f(2,3)$. Cela donne $z = 1$.

Python

- Déclaration

```
def f(x,y):
    return x**2-y
```

- Utilisation

```
# on appelle la fonction en lui passant les valeurs selon l'ordre
z = f(2, 3) # avec print(z) on trouve bien 1
# ou bien en lui passant les valeurs associées au noms des paramètres
z = f(x=2, y=3)
z = f(y=3, x=2)
```

Les paramètres sont les noms qui apparaissent entre les parenthèses lorsque l'on **définit** une fonction (dans l'exemple, x et y); les **arguments** sont les objets que l'on passe à une fonction lorsque on l'**appelle** (dans l'exemple, 2 et 3).

ATTENTION (RETURN *VS* PRINT) Ne pas confondre la fonction `print` et l'instruction `return`:

```
def f(x):
    return x**2

y = f(2)
print(f"{y = }")

y = 4
```

```
def f(x):
    print(x**2)

y = f(2)
print(f"{y = }")

4
y = None
```

Par définition de `function`, toute fonction est censée renvoyer une valeur. Une fonction qui ne renvoie pas de valeur n'est pas vraiment une fonction, mais plutôt ce que l'on appelle en programmation une procédure. En Python, même les fonctions sans instruction `return` explicite renvoient une valeur, qui est `None`. `None` représente précisément l'absence de valeur. Il sert à indiquer "il n'y a pas de valeur". C'est pourquoi on continue de les appeler fonctions, même si elles ne possèdent pas de `return` explicite.

Remarque 12 (Composition de fonctions)
On peut bien sûr composer plusieurs fonctions : soit $f, g, h, k : \mathbb{R} \to \mathbb{R}$ définies par $f(x) = x^2$, $g(x) = x+2$, $h(x) = f(g(x))$ et $k(x) = g(f(x))$. Nous écrirons

```
def f(x):
    return x**2

def g(x):
    return x+2

def h(x):
    return f(g(x))

def k(x):
    return g(f(x))

print(f(3),g(3),h(3),k(3))
```

```
9 5 25 11
```

En effet, $f(3) = 3^2 = 9$, $g(3) = 3+2 = 5$, $h(3) = f(g(3)) = f(5) = 5^2 = 25$ et $k(3) = g(f(3)) = g(9) = 9+2 = 11$.

Remarque 13 (Les tests)
Il faut toujours tester ses fonctions. Un test consiste à appeler **chaque** fonctionnalité, avec un scénario qui correspond à un cas d'utilisation, et à vérifier que cette fonctionnalité se comporte comme prévu.

En résumé :

- Une fonction en programmation peut prendre zéro, un ou plusieurs paramètres en entrée.
- Elle renvoie un ou plusieurs résultats en sortie. Si on ne renvoie pas explicitement un résultat, elle renvoie `None`.
- **Attention ! Ne confondez pas afficher et renvoyer une valeur.** L'affichage (par la commande `print()`) montre simplement quelque chose à l'écran. La plupart des fonctions renvoient une valeur (ou plusieurs) qui peut être utilisée ailleurs dans le programme (et affichée si nécessaire).
- Dès que le programme rencontre l'instruction `return`, la fonction s'arrête et renvoie le résultat. Il peut y avoir plusieurs occurrences de l'instruction `return` dans une fonction, mais seule la première rencontrée sera exécutée. On peut aussi ne pas mettre d'instruction `return` si la fonction ne renvoie rien (en réalité, elle renvoie `None`).
- Vous pouvez bien sûr faire appel à d'autres fonctions à l'intérieur d'une fonction !

6.2.1. Portée des variables

Les variables définies à l'intérieur d'une fonction **ne sont pas «visibles» depuis l'extérieur** de la fonction. On exprime cela en disant qu'une telle variable est locale à la fonction. De plus, si une variable existe déjà avant l'exécution de la fonction, tout se passe comme si, durant l'exécution de la fonction, cette variable était masquée momentanément, puis restituée à la fin de l'exécution de la fonction.

- Dans l'exemple suivant, la variable `x` est une variable locale à la fonction `f` : crée au cours de l'exécution de la fonction `f`, elle est supprimée une fois l'exécution terminée :

```
>>> def f(y):
...     x = 2
...     return 4*y
```

```
...
>>> print(f(5))
20
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

- Dans l'exemple suivant, la variable `x` est une variable qui vaut 6 à l'extérieur de la fonction et 7 au cours de l'exécution de la fonction `f`, car même si elle a un nom identique à la variable utilisée dans cette fonction, il s'agit de variables différentes pour l'interpréteur :

```
x = 6

def f(y):
    x = 7 # x est redéfinie localement, m reste inchangé
    return x*y*m

m = 2
print(f(1))
print(x)
```

```
14
6
```

- Dans cet exemple, en revanche, la variable `x` vaut 6 partout car définie à l'extérieur de la fonction et non redéfinie à l'intérieur de la fonction `f` :

```
def f(y):
    return x*y*m

x = 6
m = 2
print(f(1))
print(x)
```

```
12
6
```

On remarque que les variables utilisées dans la définition d'une fonction peuvent être affectées après la définition de la fonction mais avant de l'appeler.

Voici un exemple qui résume les différents cas :

```
def ma_fonction():
    ma_variable = 17
    s = f"Valeur de ma_variable dans ma_fonction : {ma_variable}"
    return s

ma_variable = 10
print(f"Valeur de ma_variable dans le script : {ma_variable}")
s = ma_fonction()
print(s)
print(f"Valeur de ma_variable dans le script après l'appel de ma_fonction : {ma_variable}")
```

```
Valeur de ma_variable dans le script : 10
Valeur de ma_variable dans ma_fonction : 17
Valeur de ma_variable dans le script après l'appel de ma_fonction : 10
```

Pour que la variable dans la fonction "écrase" la variable hors de la fonction (après appel de la fonction bien-sûr), il faut indiquer explicitement qu'il s'agit d'une variable globale (**déconseillé!**) :

```
def ma_fonction():
    global ma_variable
    ma_variable = 17
    s = f"Valeur de ma_variable dans ma_fonction : {ma_variable}"
    return s

ma_variable = 10
print(f"Valeur de ma_variable dans le script : {ma_variable}")
```

```
s = ma_fonction()
print(s)
print(f"Valeur de ma_variable dans le script après l'appel de ma_fonction avec global : {ma_variable}")
```

```
Valeur de ma_variable dans le script : 10
Valeur de ma_variable dans ma_fonction : 17
Valeur de ma_variable dans le script après l'appel de ma_fonction avec global : 17
```

6.2.2. Fonctions et listes

Si une liste est passée comme paramètre d'une fonction et cette fonction la modifie, cette modification se répercute sur la liste initiale. Si ce n'est pas le résultat voulu, il faut travailler sur une copie de la liste.

```
>>> def squares(a):
...     for i in range(len(a)):
...         a[i] = a[i]**2
...
>>> a = [1,2,3,4]
>>> print(a)
[1, 2, 3, 4]
>>> squares(a)
>>> print(a)
[1, 4, 9, 16]
```

Notons en passant qu'ici on n'a pas écrit d'instruction `return` : une fonction qui agit sur un objet modifiable peut se passer de l'instruction `return`.

Conseil concernant les fonctions qui modifient une liste : l'indiquer clairement, notamment en faisant en sorte que la fonction renvoie la liste modifiée :

```
def ajoute_un(liste):
    for indice in range(len(liste)):
        liste[indice] += 1
    return liste
```

```
# TEST
liste_notes = [10, 8, 16, 7, 15]
liste_notes = ajoute_un(liste_notes)
print(liste_notes)

[11, 9, 17, 8, 16]
```

Le code suivant produit la même sortie mais reste toutefois moins intuitif car il n'est pas évident de comprendre que la liste est modifiée dans la fonction :

```
def ajoute_un(liste):
    for indice in range(len(liste)):
        liste[indice] += 1
```

```
liste_notes = [10, 8, 16, 7, 15]
ajoute_un(liste_notes)
print(liste_notes)

[11, 9, 17, 8, 16]
```

6.3. Fonctions Lambda (fonctions anonymes)

Lorsque l'on utilise la syntaxe

```
def f(x, y):
    return 2*x + y
```

on crée une fonction nommée `f` qui prend deux paramètres x et y, puis on indique ce qu'elle doit faire, dans notre exemple renvoyer $2x + y$.

Il est possible de créer une fonction sans lui donner de nom, c'est ce que l'on appelle une fonction `lambda` :

$$\begin{array}{cccc} & (x,y) & \mapsto & 2x+y \\ & \downarrow & \downarrow & \downarrow \\ \texttt{lambda} & \texttt{x, y} & \texttt{:} & \texttt{2*x+y} \end{array}$$

L'écriture mathématique « $(x, y) \mapsto 2x + y$ » se lit « fonction qui à chaque couple (x, y) associe $2x + y$ ». De même, l'écriture pythonesque « `lambda x, y: 2*x+y` » se lit « fonction qui, au tuple (x, y), associe $2x + y$ ».

Déclaration avec `lambda` et utilisation.

Declaration

```
f = lambda params: vals
```

idem que

```
def f(params):
    return vals
```

Utilisation

```
res = f(args)
# ou
res = f(params=args)
```

Bien noter que **la liste des paramètres d'une fonction lambda ne s'écrit pas entre parenthèses** et que le `return` **est implicite**

Quand et comment utiliser une fonction lambda ?

- Les fonctions lambdas sont très utiles pour des fonctions courtes (et souvent temporaires). Par exemple, lorsqu'une fonction attend en paramètre une autre fonction, on peut passer une fonction anonyme sans l'affecter à une variable. Un exemple est donné à la page 270.
- On peut toujours affecter une fonction lambda à une variable. Cette deuxième utilisation est très répandue lorsque l'on doit définir une fonction très courte, souvent en combinaison avec les listes en compréhension.
  Cependant, pour éviter la tentation de coder de manière illisible, Python limite les fonctions `lambda` à **une seule ligne**. Par conséquent, pour des fonctions plus complexes ou des tâches nécessitant plusieurs lignes, il est préférable d'utiliser la syntaxe `def`.
- On peut bien sûr composer deux fonctions lambda. Par exemple, soit $f\colon x \mapsto x^2$ et $g\colon x \mapsto x-1$, et considérons h la composition de g avec f (c'est-à-dire $h(x) = g(f(x)) = g(x^2) = x^2 - 1$). On peut définir la fonction h tout naturellement comme suit :

```
>>> f = lambda x: x**2
>>> g = lambda x: x - 1
>>> h = lambda x: g(f(x))
>>> print(h(0))
-1
```

Les fonctions lambda sont largement utilisées dans les codes Python pour leur concision et leur efficacité. Elles offrent un moyen puissant de définir des fonctions, particulièrement utile dans des contextes où des fonctions temporaires et/ou simples sont nécessaires. Cependant, il est important de savoir quand les utiliser et quand préférer une définition avec `def`. En règle générale, les fonctions lambda sont idéales pour des fonctions courtes. Cependant, dès que la logique devient complexe ou nécessite plusieurs lignes de code, une définition avec `def` devient souvent plus appropriée. Pour choisir entre les deux, la priorité doit être accordée à la lisibilité du code : une fonction lambda complexe peut rendre le code difficile à comprendre. En contrepartie, une fonction très courte définie avec lambda peut souvent offrir une compréhension plus immédiate de l'algorithme.

6.4. ☆ Indication des types des variables à l'entrée et à la sortie des fonctions

Depuis Python 3.6, il est possible d'indiquer les types des variables à l'entrée et à la sortie des fonctions. C'est une fonctionnalité utile pour indiquer à qui doit l'utiliser ce que vos fonctions attendent en entrée et en sortie.

Syntaxe :

```
def nom_de_fonction(var1: type1, var2: type2...) -> type_sortie
    ...
    return sortie
```

Exemples :

```
def afficher(chanson: str) -> None:
    print(chanson)
    return None

def calculer_moyenne_3(a: float, b: float, c: float) -> float:
    moy = (a+b+c)/3
    return moy

def calculer_moyenne(tableau: list[float]) -> float:
```

```
    moy = sum(L)/len(L)
    return moy
```

6.5. ☆ Arguments par défaut

Nous avons déjà vu que les fonctions prédéfinies peuvent avoir des valeurs par défaut pour certains paramètres. Ces valeurs par défaut permettent aux utilisateurs d'appeler la fonction sans spécifier les arguments correspondant aux paramètres disposant de valeurs par défaut. Toutefois, si une valeur est explicitement passée, elle prendra le pas sur la valeur par défaut définie.

Il n'est pas nécessaire de donner des valeurs par défaut à tous les paramètres. Dans ce cas, il est important de respecter un ordre précis lors de la déclaration des paramètres : placer d'abord les paramètres sans valeur par défaut, suivis des paramètres ayant des valeurs par défaut. De cette manière, les arguments fournis lors de l'appel de la fonction remplaceront en priorité les paramètres sans valeur par défaut.

```
# Déclaration d'une fonction avec :
# - un argument sans valeur par défaut : nom
# - un argument avec une valeur par défaut : prenom
def bonjour(nom, prenom="Gloria"):
    return f"Bonjour {prenom} {nom.upper()} !"
```

Lors de l'appel de cette fonction, il faudra toujours passer au moins une valeur pour le paramètre `nom`.

Le même principe pour l'ordre s'applique lors de l'invocation de la fonction :

```
print("Cas 1 PP : ", bonjour("Faccanoni", "Daniela"))  # avec des paramètres positionnels
print("Cas 2 PN : ", bonjour("Faccanoni", prenom="Daniela"))  # avec un paramètre positionnel et un
↪ paramètre nommé
print("Cas 3 NN : ", bonjour(prenom="Daniela",nom="Faccanoni"))  # avec deux paramètres nommés, ordre
↪ non important
print("Cas 4 PD: ", bonjour("Faccanoni"))  # avec un seul paramètre positionnel, l'autre prenant la
↪ valeur par défaut
print("Cas 5 ND: ", bonjour(nom="Faccanoni"))  # avec un seul paramètre nommé, l'autre prenant la valeur
↪ par défaut
```

```
Cas 1 PP :  Bonjour Daniela FACCANONI !
Cas 2 PN :  Bonjour Daniela FACCANONI !
Cas 3 NN :  Bonjour Daniela FACCANONI !
Cas 4 PD:  Bonjour Gloria FACCANONI !
Cas 5 ND:  Bonjour Gloria FACCANONI !
```

6.6. ☆ Fonctions récursives

Une fonction récursive est tous simplement une fonction qui s'appelle elle même lors de son exécution. Lorsqu'on définit une fonction récursive, il faudra toujours faire bien attention à fournir une condition qui sera fausse à un moment ou l'autre au risque que la fonction s'appelle à l'infini.

Une fonction mathématique définie par une relation de récurrence et une condition initiale peut être programmée de manière récursive de façon naturelle. Par exemple :

$$\begin{cases} u_0 = 1, \\ u_{n+1} = 2u_n \end{cases}$$

```
#Version recursive
def UR(n):
    if n==0: # condition de sortie
        return 1
    else:
        return 2*UR(n-1)

print(UR(6))
```

```
64
```

Parfois on peut expliciter la récursion. Dans notre exemple nous avons

$$u_{n+1} = 2u_n = 2^2 u_{n-1} = \cdots = 2^{n+1} u_0,$$

ainsi $u_6 = 2^6 \times 1 = 64$ que l'on peut implémenter comme suit :

```
#Version itérative
def UI(u):
    return 2*u

N,u = 5,1
for n in range(N+1):
    u = UI(u)
print(u)
```

```
64
```

```
#Version explicite
def UE(u0,n):
    return 2**n*u0

N, u0 = 5, 1
u = UE(u0,N+1)
print(u)
```

```
64
```

Prenons un autre exemple : le calcul de la somme des entiers entre 1 et n. L'idée est d'expliquer comment il faut commencer et ce qu'il faut faire pour passer de l'étape $n-1$ à l'étape n. Pour commencer, si n vaut 1, la somme vaut 1. Ensuite si on a déjà calculé la somme de 1 à $n-1$, il suffit de lui ajouter n pour obtenir la somme de 1 à n. Ici aussi on peut expliciter la récursion :

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

ainsi $1+2+3+\cdots+10 = \frac{10\times 11}{2} = 55$.

Version récursive :

```
def somme(n):
    if n==1:
        return 1
    else:
        return somme(n-1)+n

print(somme(10))
```

```
55
```

Version itérative :

```
N=10
somme=0
for n in range(1,N+1):
    somme+=n
print(somme)
```

```
55
```

Version explicite :

```
N=10
somme=N*(N+1)/2
print(somme)
```

```
55.0
```

Attention :

- ne pas oublier d'initialiser c'est à dire d'expliquer ce que doit faire le programme pour les valeurs initiales. Si on n'explique pas dans notre exemple quoi faire si $n = 1$ alors le programme va chercher à calculer `somme(1)` puis `somme(0)` puis `somme(-1)` sans jamais s'arrêter ;
- ne pas demander des calculs trop complexes. L'avantage d'une écriture récursive est que c'est en général simple de notre coté mais pas forcément pour l'ordinateur. Cela veut dire qu'il faut quand même se demander si ce qu'on lui demande ne va pas être trop complexe et s'il n'y a pas des moyens de lui simplifier la tache ;
- en python, de base, on ne peut faire que 1000 appels de la même fonction de façon récursive (c'est-à-dire que, dans notre exemple, on ne pourra pas calculer `somme(1001)`.

6.7. Exercices

6.7.1. Exercices spécifiques aux fonctions

Exercice 6.1 (Devine le résultat - bêtisier)
Voici un bêtisier pour mieux comprendre les règles. Proposer une correction.

Cas 1 :
```
def f(x)
    return 2*x+1
```

Cas 2 :
```
def f(x):
return 2*x+1
```

Cas 3 :
```
def f(x):
    2*x+1

print(f(2))
```

Cas 4 :
```
def f(x):
    a = 2*x+1
    return a
    print('Hello')

print(f(2))
```

Correction

Cas 1 : il manque les deux-points en fin de ligne :
```
>>> def f(x)
  File "<stdin>", line 1
    def f(x)
SyntaxError: expected ':'
>>> ⟶return 2*x+1
  File "<stdin>", line 1
    ⟶return 2*x+1
    ^
IndentationError: unexpected indent
```

Cas 2 : il manque l'indentation :
```
>>> def f(x):
... return 2*x+1
  File "<stdin>", line 2
    return 2*x+1
    ^^^^^^
IndentationError: expected an indented block after function definition on line 1
```

Cas 3 : il manque le mot `return` et donc tout appel de la fonction aura comme réponse `None` :
```
>>> def f(x):
... ⟶2*x+1
...
>>> print(f(2))
None
```
Cette fonction exécute le calcul mais elle ne renvoie pas d'information spécifique. Pour qu'une fonction renvoie une certaine valeur, il faut utiliser le mot-clé `return`.

Cas 4 : l'instruction `print('Hello')` n'est jamais lue par Python car elle apparaît après l'instruction `return` :
```
>>> def f(x):
... ⟶a = 2*x+1
... ⟶return a
... ⟶print('Hello')
...
>>> print(f(2))
5
```

Exercice 6.2 (Devine le résultat - variables globales *vs* locales)
Cet exercice traite de la portée des variables en Python. En termes simples, la portée d'une variable indique les endroits dans le script où elle peut être utilisée. Une variable peut avoir une portée locale ou une portée globale.

Les variables définies à l'intérieur d'une fonction sont appelées variables locales. Elles ne peuvent être utilisées que localement, c'est-à-dire uniquement à l'intérieur de la fonction qui les a définies. Tenter accéder à une variable locale en dehors de la fonction provoquera une erreur. Chaque fois qu'une fonction est appelée, Python crée un nouvel espace mémoire pour elle, où sont stockées les variables locales. Cet espace est inaccessible depuis l'extérieur de la fonction et est automatiquement détruit une fois que la fonction a terminé son exécution.

En revanche, les variables définies en dehors de toute fonction, c'est-à-dire dans l'espace global du script, sont

appelées variables globales. Elles sont accessibles dans l'ensemble du script, mais ne peuvent être modifiées à l'intérieur des fonctions. Autrement dit, une fonction peut utiliser la valeur d'une variable globale, mais ne peut pas la redéfinir. Si on tente de redéfinir une variable globale à l'intérieur d'une fonction, cela créera simplement une nouvelle variable locale avec le même nom, distincte de la variable globale d'origine (cette observation ne s'applique plus lorsqu'on travaille avec des objets mutables tels que les listes).

Prévoir le résultat et, si un script provoque une erreur, expliquer pourquoi.

Cas 1 :
```
def test():
    x = "hello"

test()
print(x)
```

Cas 2 :
```
def test():
    print(x)

x = "hello"
test()
```

Cas 3 :
```
def test():
    x="ciao"
    print(x)

x = "hello"
test()
print(x)
```

Correction

Cas 1 : Une variable déclarée dans une fonction ne sera visible que dans cette fonction. On parle alors de variable locale :

```
>>> def test():
...     x = "hello"
...
>>> test()
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Cas 2 : Une variable déclarée en dehors d'une fonction est visible à l'intérieur de la fonction. On parle alors de variable globale.

```
>>> def test():
...     print(x)
...
>>> x = "hello"
>>> test()
hello
```

Cas 3 : Une variable globale (donc déclarée en dehors d'une fonction) peut être redéfinie à l'intérieur de la fonction mais elle reprendra sa valeur globale en dehors.

```
>>> def test():
...     x="ciao"
...     print(x)
...
>>> x = "hello"
>>> test()
ciao
>>> print(x)
hello
```

Remarque : dans certaines situations, il serait utile de pouvoir modifier la valeur d'une variable globale depuis une fonction, notamment dans le cas où une fonction se sert d'une variable globale et la manipule. Pour faire cela, il suffit d'utiliser le mot clef `global` devant le nom d'une variable globale utilisée localement afin d'indiquer à Python qu'on souhaite bien modifier le contenu de la variable globale et non pas créer une variable locale de même nom.

```
>>> def test():
...     global x
...     x="ciao"
...     print(x)
...
>>> x = "hello"
>>> test()
ciao
>>> print(x)
ciao
```

Exercice Bonus 6.3 (Devine le résultat - Function scope)

Étudier le code ci-dessous et prévoir le résultat avant de l'exécuter.

```
def outer_func():
    def inner_func():
        a = 9
        print(f'inside inner_func, a is {a:d} (id={id(a):d})')
        print(f'inside inner_func, b is {b:d} (id={id(b):d})')
        print(f'inside inner_func, len is {len:d} (id={id(len):d})')

    len = 2
    print(f'inside outer_func, a is {a:d} (id={id(a):d})')
    print(f'inside outer_func, b is {b:d} (id={id(b):d})')
    print(f'inside outer_func, len is {len:d} (id={id(len):d})')
    inner_func()

a, b = 6, 7
outer_func()
print(f'in global scope, a is {a:d} (id={id(a):d})')
print(f'in global scope, b is {b:d} (id={id(b):d})')
print(f'in global scope, len is {len} (id={id(len):d})')
```

Source : https://scipython.com/book/chapter-2-the-core-python-language-i/examples/function-scope/

Correction

```
inside outer_func, a is 6 (id=11760840)
inside outer_func, b is 7 (id=11760872)
inside outer_func, len is 2 (id=11760712)
inside inner_func, a is 9 (id=11760936)
inside inner_func, b is 7 (id=11760872)
inside inner_func, len is 2 (id=11760712)
in global scope, a is 6 (id=11760840)
in global scope, b is 7 (id=11760872)
in global scope, len is <built-in function len> (id=123205974581840)
```

Ce programme définit une fonction, `inner_func`, imbriquée dans une autre, `outer_func`. Après ces définitions, l'exécution se déroule comme suit :

1. Les variables globales `a=6` et `b=7` sont initialisées.
2. `outer_func` est appelée :
   - 2.1. `outer_func` définit une variable locale, `len=2`.
   - 2.2. Les valeurs de `a` et `b` sont affichées ; elles n'existent ni localement ni dans la fonction englobante, de sorte que Python les cherche et les trouve globalement : leurs valeurs (6 et 7) sont affichées.
   - 2.3. La valeur de la variable locale `len` (2) est affichée.
   - 2.4. `inner_func` est appelé :
     - 2.4.1. Une variable locale, `a=9`, est définie.
     - 2.4.2. La valeur de cette variable locale est affichée.
     - 2.4.3. La valeur de `b` est affichée ; `b` n'existe pas localement, alors Python la cherche dans la fonction englobante `outer_func`. Elle n'y est pas trouvée non plus, alors Python procède à une recherche globale où elle est trouvée : la valeur `b=7` est affichée.
     - 2.4.4. La valeur de `len` est affichée : `len` n'existe pas localement, mais elle se trouve dans la fonction englobante puisque `len=2` est définie dans `outer_func` : sa valeur est affichée.
3. Une fois l'exécution de `outer_func` terminée, les valeurs de `a` et `b` globales sont affichées.
4. La valeur de `len` est affichée. Cette valeur n'est pas définie globalement, de sorte que Python recherche ses propres noms de fonctions built-in : `len` est la fonction intégrée permettant de déterminer la longueur des séquences. Cette fonction est elle-même un objet et elle fournit une brève description d'elle-même sous forme de chaîne de caractères lorsqu'elle est affichée.

Bien noter que dans `outer_func` le nom `len` est l'objet entier 2. Cela signifie que la fonction intégrée `len` d'origine n'est pas disponible dans cette fonction (et qu'elle n'est pas non plus disponible dans la fonction incluse, `inner_func`).

Exercice 6.4 (Devine le résultat)

Soit la fonction suivante définie en Python

```
def f(x,a,b):
    if a>b:
        a,b=b,a
    if x<=a:
        return a
    elif x>=b:
        return b
    else:
        if (x-a)>(b-x):
            return b
        else:
            return a
```

Calculez les valeurs de la fonction pour les arguments suivants: $f(0,0,0)$, $f(-2,0,3)$, $f(-2,3,0)$, $f(2,0,2)$, $f(1,0,2)$, $f(3,-1,-2)$.

Correction

La fonction renvoie les résultats suivants: $f(0,0,0) = 0$ $f(-2,0,3) = 0$ $f(-2,3,0) = 0$ $f(2,0,2) = 2$ $f(1,0,2) = 0$ $f(3,-1,-2) = -1$

Nous pouvons reformuler cette fonction en termes mathématiques comme suit:

$$f(x,a,b) = \begin{cases} f(x,b,a) & \text{si } a > b \\ a & \text{si } x \leq a \text{ avec } a \leq b \\ b & \text{si } x \geq a \text{ avec } a \leq b \\ b & \text{si } (x-a) > (b-x) \text{ avec } a \leq b \\ a & \text{sinon} \end{cases}$$

On remarque que, si $a > b$, nous pouvons échanger a et b: nous pouvons donc restreindre l'étude au cas $a \leq b$ comme suit:

$$f(x,a,b) = \begin{cases} g(x,b,a) & \text{si } a > b \\ g(x,a,b) & \text{sinon} \end{cases} \qquad \text{où} \qquad g(x,a,b) = \begin{cases} a & \text{si } x \leq a \text{ ou si } (x-a) \leq (b-x) \\ b & \text{si } x \geq a \text{ ou si } (x-a) > (b-x) \end{cases} = \begin{cases} a & \text{si } x \leq \frac{a+b}{2} \\ b & \text{si } x > \frac{a+b}{2} \end{cases}$$

Nous trouvons alors les valeurs suivantes pour les arguments donnés:

$$f(0,0,0) = 0, \qquad f(-2,0,3) = 0, \qquad f(-2,3,0) = f(-2,0,3) = 0,$$
$$f(2,0,2) = 2, \qquad f(1,0,2) = 0, \qquad f(3,-1,-2) = f(3,-2,-1) = -1.$$

Exercice 6.5 (Renvoyer un booléen)

L'utilisation d'une instruction `if` pour renvoyer un booléen ou pour définir une variable comme booléenne est redondante. Réécrire la fonction suivante de façon plus compacte et lisible:

```
def legal_age(age):
    # float -> bool
    if age >= 18:
        return True
    else:
        return False
```

Correction

Une fonction qui renvoie le booléen `True` si l'âge d'une personne est égal ou supérieur à 18 ans et `False` dans le cas contraire, pourrait s'écrire comme la fonction donnée. Cependant, `age >= 18` donnera déjà un booléen. Cela signifie que la fonction peut être écrite de manière beaucoup plus simple et plus propre comme suit:

```
def legal_age(age):
    return age >= 18
```

Exercice 6.6 (F: $\mathbb{R} \to \mathbb{R}^2$)

Devine les résultats :

```
f1 = lambda x:x+1
f2 = lambda x:x**2
F = [f1,f2]
print( [f(1) for f in F] )

F = lambda x : [x+1,x**2]
print( F(1) )

F = lambda x : [f1(x),f2(x)]
print( F(1) )
```

Correction

$$\begin{aligned} F : \mathbb{R} &\to \mathbb{R}^2 \\ x &\mapsto \begin{pmatrix} f_1(x) \\ f_2(x) \end{pmatrix} = \begin{pmatrix} x+1 \\ x^2 \end{pmatrix} \end{aligned}$$

On obtient

```
[2, 1]
[2, 1]
[2, 1]
```

Exercice Bonus 6.7 (Arguments nommés et arguments par défaut)

Devine les résultat :

```
x0, k0 = 10, 3

# Sans valeur par défaut
def f(x, k): return x**k

f(x0, k0)       # 10^3
f(k0, x0)       # 3^10
f(x=x0, k=k0)   # 10^3
f(k=k0, x=x0)   # 10^3
f(x0, k=k0)     # 10^3
f(x=x0, k0)     # ERREUR
```

```
# Avec 1 valeur par défaut
def g(x, k=2): return x**k

g(x0, k0)       # 10^3
g(x=x0, k=k0)   # 10^3
g(k=k0, x=x0)   # 10^3
g(x0, k=k0)     # 10^3
g(x=x0, k0)     # ERREUR

g(x0)           # 10^2
g(x=x0)         # 10^2
g(k0)           # 3^2
g(k=k0)         # ERREUR
```

```
# Définition incorrecte
def h(x=10, k): return x**k
```

Correction

Les fonctions f et g calculent x^k. La différence dans la définition : g fournit une valeur par défaut pour k. Une valeur par défaut ne peut être suivie d'un paramètre sans valeur par défaut, d'où l'erreur dans la définition de h.

Attention aux différents types d'appels : par position (l'ordre est important) ou par nom.

```
>>> x0, k0 = 10, 3

>>> def f(x, k): return x**k
...
>>> print(f"x0={x0}, k0={k0}")
x0=10, k0=3
>>> print(f"{f(x0,k0) = }   {f(k0,x0) = }   {f(x=x0,k=k0) = }")
f(x0,k0) = 1000   f(k0,x0) = 59049   f(x=x0,k=k0) = 1000

>>> def g(x, k=2): return x**k
...
>>> print(f"{g(x0,k0) = }   {g(x0) = }   {g(k0) = }")
g(x0,k0) = 1000   g(x0) = 100   g(k0) = 9
```

Démarche générale

- **Compréhension de l'énoncé.** Commencez par lire attentivement l'énoncé pour bien comprendre les exigences. Identifiez les éléments à traiter et l'objectif de la fonction à développer. Prenez des notes sur papier si nécessaire et reformulez les consignes en français.
- **Identification des entrées et sorties.**
  - Inputs : identifiez les paramètres ou les données d'entrée requis par la fonction. Précisez leur type (entiers, chaînes de caractères, listes, etc.).
  - Outputs : déterminez le type de données que la fonction doit renvoyer en sortie.
- **Écriture d'un canevas de la fonction et des tests.** Créez une structure tenant compte de la signature de la fonction (nom, inputs et outputs). Utilisez un canevas avec éventuellement des commentaires pour décrire les étapes intermédiaires. Voici un exemple :

```
# ==========================================
def nom_fct(parametres):
    # Instructions pour résoudre le problème
    # ...
    return sorties

# TESTS
# ==========================================

# TEST 1
entree = ...  # Entrée de test, vérifier le type
attendu = ... # Résultat connu attendu
output = nom_fct(entree)  # Résultat obtenu
print( f"nom_fct({entree}) = {output}, on attendait {attendu}" )

# TEST 2
...
```

- **Implémentation du corps de la fonction.**
  - Une fois la structure en place, écrivez le code nécessaire pour résoudre le problème. Cela peut impliquer l'utilisation de boucles, de conditions, de calculs, etc.
  - Assurez-vous de prendre en compte toutes les contraintes et règles énoncées dans la description du problème.
- **Test de la fonction.** Testez la fonction avec différentes entrées pour garantir son bon fonctionnement. En cas d'échec, examinez attentivement le message d'erreur généré par Python pour identifier la source du problème (le numéro de la ligne qui pose problème est aussi indiqué). Effectuez des tests simples avec des résultats attendus connus pour vérifier l'exactitude de la fonction.

  Bien entendu, la réussite des tests ne garantit pas l'exactitude totale de la fonction, mais plutôt qu'elle présente un comportement approprié pour les scénarios de test spécifiques. Par conséquent, il faut concevoir des tests qui couvrent tous les cas particuliers et les scénarios spécifiques afin d'assurer une validation la plus complète possible de la fonction.
- **LE SILENCE EST D'OR !** Dans chaque exercice, veillez à développer des fonctions qui ne produisent aucune sortie visible, mais qui renvoient silencieusement le résultat attendu, même s'il s'agit d'une chaîne de caractères (à moins d'indication contraire dans l'énoncé de l'exercice). L'utilisation de `print` est autorisée pour vérifier le résultat renvoyé par la fonction, mais seulement à des fins de débogage pendant la phase de développement. N'oubliez pas de commenter ces instructions `print` à la fin.

Exercice 6.8 (Calcul de l'effectif et de la fréquence – démarche guidée)

On souhaite écrire une fonction `frequence_element(L, n)` qui renvoie l'effectif et la fréquence d'un élément `n` dans une liste `L`. Suivez la démarche pas à pas.

1. **Compréhension de l'énoncé.**
   - Que représente `L` ?
   - Que représente `n` ?

- Quelles sont les valeurs attendues en sortie ?

2. **Identification des entrées et sorties.**
   - Quels sont les **types** des entrées et sorties de la fonction ?
   - Quelle sera la structure de la sortie (tuple, liste, entier, float, str, ...) ?
3. **Canevas de la fonction.** Écrivez le squelette suivant :

```python
def frequence_element(L, n):
    effectif = ...    # 1. Calculer l'effectif de n dans L
    frequence = ...   # 2. Calculer la fréquence de n dans L
    return ...        # 3. Retourner les résultats
```

4. **Tests de la fonction.** Avant de compléter la fonction, réfléchissez : quels tests simples pouvez-vous effectuer pour vérifier le bon fonctionnement de la fonction ?
5. **Implémentation.** Remplissez les parties manquantes (...) dans le canevas de la fonction.
6. **Vérification.** Testez la fonction avec différentes valeurs de `n` et différentes listes `L`. Observez si les résultats correspondent à vos attentes.

Correction

1. **Compréhension de l'énoncé.** `L` est une liste d'entiers, `n` est un entier dont on veut calculer l'effectif et la fréquence. La fonction doit renvoyer deux valeurs : le nombre de fois que `n` apparaît dans `L` (effectif) et la fréquence correspondante (effectif divisé par la longueur de `L`).
2. **Identification des entrées et sorties**
   - Entrées :
     `L : list[int]`
     `n : int`
   - Sorties :
     `tuple[int, float] -> (effectif, fréquence)`
4. **Tests de la fonction.** Choisissons `L = [0, 10, 20, 10, 20, 30, 20, 30, 40, 20]`.
   - Si $n = 20$, l'effectif est 4 (il apparaît 4 fois dans la liste) et la fréquence est $4/10 = 0.4$ ainsi le résultat attendu est $(4, 0.4)$;
   - si $n = 10$, l'effectif est 2 et la fréquence = 2 / 10 = 0.2 ainsi le résultat attendu est (2, 0.2) ;
   - si $n = 5$, l'effectif est 0 (il n'apparaît pas dans `L`) et la fréquence = 0 / 10 = 0 ainsi le résultat attendu est (0, 0.0).
5. **Implémentation.**

```python
def frequence_element(L, n):
    effectif = L.count(n)
    frequence = effectif / len(L)
    return (effectif, frequence)
```

6. **Vérification.**

```python
L = [0, 10, 20, 10, 20, 30, 20, 30, 40, 20]

# Test 1
n = 20
print(frequence_element(L, n))  # (4, 0.4)

# Test 2
n = 10
print(frequence_element(L, n))  # (2, 0.2)

# Test 3
n = 5
print(frequence_element(L, n))  # (0, 0.0)
```

```
(4, 0.4)
(2, 0.2)
(0, 0.0)
```

Exercice 6.9 (CodEx: Rendement d'un livret – démarche guidée)

Tom a ouvert un livret dans une banque, qui lui propose un rendement de 3.5% par an. Il y a placé la somme de 1000 € et se demande maintenant en combien d'années cette somme sera doublée.

On souhaite écrire la fonction `double_placement`. Cette fonction prend en paramètres un nombre strictement positif `placement` qui représente la somme placée, et un nombre `rendement`, compris entre 0 et 100, qui correspond au rendement exprimé en pourcents. Cette fonction renvoie un entier qui correspond au nombre d'années nécessaires pour le doublement du placement initial.

Suivez la démarche pas à pas.

1. **Compréhension de l'énoncé.**
   - Que représente `placement` ?
   - Que représente `rendement` ?
   - Que représente la valeur renvoyée par la fonction ?
2. **Identification des entrées et sorties.**
   - Quels sont les types des entrées ?
   - Quel est le type de la sortie ?
3. **Modélisation mathématique.**
   - Exprimer le capital après n années en fonction du placement initial et du rendement.
   - Traduire mathématiquement la condition de doublement du capital.
4. **Canevas de la fonction.**
```
def double_placement(placement, rendement):
    capital = ...
    annees = ...
    ...
    return ...
```
5. **Tests de la fonction.**
   - Que doit renvoyer la fonction pour `placement = 1000` et `rendement = 3.5` ?
   - Comparer le résultat obtenu avec la valeur théorique donnée par la formule
$$n \geq \frac{\ln(2)}{\ln\left(1 + \frac{\texttt{rendement}}{100}\right)}.$$
6. **Implémentation.** Compléter le canevas de la fonction.
7. **Vérification.** Tester la fonction avec différentes valeurs de `placement` et de `rendement`.

Source : https://codex.forge.apps.education.fr/exercices/rendement_livret/

Correction

1. **Compréhension de l'énoncé**. `placement` est la somme initialement placée (en euros). `rendement` est le taux d'intérêt annuel exprimé en pourcentage. Chaque année, la somme placée augmente selon ce rendement, c'est-à-dire que si le capital vaut C, alors après une année il devient $C \times \left(1 + \frac{\texttt{rendement}}{100}\right)$. La fonction doit calculer le nombre d'années nécessaires pour que la somme devienne au moins le double de la somme initiale.
2. **Identification des entrées et sorties**
   - Entrées :
```
placement : float
rendement : float
```
   - Sortie :
```
int -> nombre d'années
```
3. **Canevas de la fonction.**
```
def double_placement(placement, rendement):
    capital = ...    # capital actuel
    annees = ...     # compteur d'années
    ...
    return ...
```
4. **Tests de la fonction.** On choisit `placement = 1000` et `rendement = 3.5`. Pour vérifier le résultat, on peut calculer la valeur théorique. Après n années, le capital initial C_0 devient
$$C_n = C_0 \times \left(1 + \frac{\texttt{rendement}}{100}\right)^n.$$

Le capital est doublé lorsque $C_n \geq 2C_0$, ce qui revient à résoudre l'inéquation

$$\left(1 + \frac{\texttt{rendement}}{100}\right)^n \geq 2.$$

En prenant le logarithme, on obtient

$$n \geq \frac{\ln(2)}{\ln\left(1 + \frac{\texttt{rendement}}{100}\right)} \approx 20.1.$$

On s'attend donc à ce que la fonction renvoie 21 années.

5. **Implémentation.**

```python
def double_placement(placement, rendement):
    capital = placement
    annees = 0
    while capital < 2 * placement:
        capital = capital * (1 + rendement / 100)
        annees += 1
    return annees
```

6. **Vérification.**

```python
print(double_placement(1000, 3.5))
```

```
21
```

La fonction renvoie le nombre d'années nécessaires pour que la somme placée atteigne ou dépasse le double du placement initial.

Exercice 6.10 (Mes premières potions magiques avec `def`)

Tu es un apprenti sorcier et tu vas créer tes premières potions magiques. Chaque potion est représentée par une fonction. Suis bien ces consignes :

- vérifie si la potion (=fonction) a besoin d'ingrédients (paramètres d'entrée),
- vérifie si elle doit produire un résultat (qu'elle renverra avec `return`),
- valide ta potion avec des tests bien choisis (ne mets pas de `print` à l'intérieur de la potion mais seulement dans les tests après avoir récupéré le résultat).

Résumé des potions magiques : nom, entrée et sortie

Nom de la fonction (nom de la Potion)	**Entrée (Ingrédients)**	**Sortie (Effet)**
`my_PI`	aucune	renvoie une valeur (3.1415)
`bonbon2poison`	1 paramètre (bonbons)	une valeur (1.08× bonbons)
`pouvoir_magique`	2 paramètres (x,n)	une valeur (x^n)
`somme_et_potion`	2 paramètres (a,b)	deux valeurs ($a+b$ et ab)
`ajouter_mandragore`	1 paramètre (liste)	aucun retour, modifie la liste

Potions sans ingrédients mais qui donnent un résultat

- Crée la potion `my_PI()` qui renvoie la poudre magique spéciale `3.1415`.

Potions avec ingrédients et résultat

- Crée la potion `bonbon2poison(bonbons)` : convertit un nombre de bonbons en unités de poison (1 bonbon = 1.08 unités de poison).
- Crée la potion `pouvoir_magique(x,n)` : calcule la puissance magique d'un ingrédient selon la formule x^n.
- Crée la potion `somme_et_potion(a,b)` : calcule la somme et le produit magique de deux ingrédients.

Potions avec un ingrédient mais sans retour

- Soit R une liste. Crée la potion `ajouter_mandragore(R)` : ajoute la `"mandragore"` à la fin de la liste R sans renvoyer aucun objet.

Correction

Voici comment préparer tes potions :

Potion magique : ~~sans ingrédients~~ mais avec résultat

```python
def my_PI():
    return 3.1415
```

```python
# TEST
resultat = my_PI()
print(f"La potion renvoie {resultat} unités de
↪  poudre magique")
```

```
La potion renvoie 3.1415 unités de poudre magique
```

◇

Potion magique : ingrédients et résultat

```python
def bonbon2poison(bonbons):
    return bonbons * 1.08
```

```python
# TEST
bonbons = 10
poison = bonbon2poison(bonbons)
print(f"{bonbons} bonbons = {poison:.2f} unité(s)
↪  de poison")
```

```
10 bonbons = 10.80 unité(s) de poison
```

◇

Potion magique : ingrédients et résultat

```python
def pouvoir_magique(x, n):
    return x**n
```

```python
# TEST
p = pouvoir_magique(2,10)
print(f"La puissance magique est {p}")
```

```
La puissance magique est 1024
```

◇

Potion magique : ingrédients et résultat

```python
def somme_et_potion(a,b):
    return a+b, a*b
```

```python
# TEST
x, y = 6,7
s, p = somme_et_potion(x,y)
print(f"{x}+{y}={s}, {x}*{y}={p}")
```

```
6+7=13, 6*7=42
```

◇

Potion magique : ingrédients ~~sans résultat~~

```python
def ajouter_mandragore(L):
    L.append("mandragore")
```

```python
# TEST
recette = ["dragonnet", "phénix"]
print(f"{recette = } avant appel")
ajouter_mandragore(recette)
print(f"{recette = } après appel")
```

```
recette = ['dragonnet', 'phénix'] avant appel
recette = ['dragonnet', 'phénix', 'mandragore']
↪  après appel
```

Correspondance entre fonctions mathématiques et fonctions Python

Ce paragraphe a pour but d'illustrer, à travers quelques exemples simples, la correspondance entre l'écriture mathématique d'une fonction et sa traduction en langage Python. Chaque exemple présente :

- la **définition** de la fonction sous forme mathématique, puis sous les deux formes possibles en Python : avec l'instruction `def` et avec une *lambda-fonction* ;
- une **affectation**, c'est-à-dire l'appel de la fonction sur une ou plusieurs valeurs ;
- et enfin le **résultat** obtenu.

Ces tableaux permettent ainsi de faire le lien entre la logique mathématique et sa mise en œuvre concrète dans un code Python.

	Math	Python (def)	Python (lambda)
Définition	$f : x \mapsto x^2$	`def f(x):` `  return x**2`	`f = lambda x: x**2`
Affectation	$y = f(2)$	`y = f(2)` ou `y = f(x=2)`	
Résultat	$y = 4$	`y = 4`	

	Math	Python (def)	Python (lambda)
Définition	$(x, y) \mapsto x + y$	`def f(x, y):` `  return x+y`	`f = lambda x, y: x+y`
Affectation	$z = f(2,3)$	`z = f(2, 3)` ou `z = f(x=2, y=3)`	
Résultat	$z = 5$	`z = 5`	

	Math	Python (def)	Python (lambda)
Définition	$(x, y) \mapsto (x + y, x - y)$	`def f(x, y):` `  return (x+y, x-y)`	`f = lambda x, y: (x+y, x-y)`
Affectation	$(u, v) = f(2,3)$	`u, v = f(2, 3)` ou `u, v = f(x=2, y=3)`	
Résultat	$(u, v) = (5, -1)$	`u` ← `5` et `v` ← `-1`	

	Math	Python (def)	Python (lambda)

Définition	$f(x) = \begin{cases} 2 & x \geq 0 \\ -x & x < 0 \end{cases}$	`def f(x):`
		`  if x >= 0:`
		`  return 2`
		`  else:`
		`  return -x`
			`f = lambda x: 2 if x>=0 else -x`

Affectation	$y = f(-3)$	`y = f(-3)` ou `y = f(x=-3)`	
Résultat	$y = 3$	`y` ← `3`	

Exercice 6.11 (Mes premières fonctions λ)

Écrire les fonctions suivantes d'abord comme une fonction `lambda` puis avec le mot clé `def`. Vérifier leur définition sur des tests bien choisis.

$$f_1 : \mathbb{R} \to \mathbb{R} \qquad x \mapsto x^2 + 5$$

$$f_2 : \mathbb{R}^2 \to \mathbb{R} \qquad (x, y) \mapsto x + y - 2$$

$$f_3 : \mathbb{R} \to \mathbb{R}^2 \qquad x \mapsto (x^2, x^3)$$

$$f_4 : \mathbb{R}^2 \to \mathbb{R}^2 \qquad (x, y) \mapsto (x + y, x - y)$$

$$f_5 : \mathbb{R} \to \mathbb{R} \qquad x \mapsto \begin{cases} 3 & \text{si } x < 10 \\ -2x & \text{si } 10 \leq x < 15 \\ x^2 & \text{sinon} \end{cases}$$

Correction

- Avec des fonctions lambda :

①
```
f1 = lambda x : x**2+5
print( f1(2) )
```
```
9
```

②
```
f2    = lambda x,y : x+y-2 # pas de parenthèses ou crochets pour les entrées
f2bis = lambda t : t[0]+t[1]-2 # l'entrée est un tuple ou une liste (ou un string)
print( f2(2,3) )
print( f2bis((2,3)) ) # noter ((...)) : l'argument est un tuple
print( f2bis([2,3]) ) # noter ([...]) : l'argument est une liste
```

```
3
3
3
```

③
```
f3    = lambda x : (x**2,x**3) # on peut oublier les () car on a déjà un tuple grâce à la virgule
f3bis = lambda x : [x**2,x**3] # ne pas oublier les [] si on veut comme résultat une liste
print( f3(2) )
print( f3bis(2) )
```

```
(4, 8)
[4, 8]
```

④
```
f4        = lambda x,y : (x+y,x-y)
f4bis     = lambda x,y : [x+y,x-y]
f4ter     = lambda t : (t[0]+t[1],t[0]-t[1])
f4quatuor = lambda t : [t[0]+t[1],t[0]-t[1]]
print( f4(2,3) )
print( f4bis(2,3) )
print( f4ter((2,3)) )
print( f4ter([2,3]) )
print( f4quatuor((2,3)) )
print( f4quatuor([2,3]) )
```

```
(5, -1)
[5, -1]
(5, -1)
(5, -1)
[5, -1]
[5, -1]
```

⑤
```
f5    = lambda x : 3 if (x<10) else ( -2*x if (10<=x<15) else x**2 )
f5bis = lambda x : 3*(x<10) - 2*x*(10<=x<15) + x**2*(x>=15)
print( f5(2), f5(12), f5(22) )
print( f5bis(2), f5bis(12), f5bis(22) )
```

```
3 -24 484
3 -24 484
```

Remarquons que ces deux versions du code ne sont pas équivalentes. Dans la fonction `f5bis`, tous les tests sont évalués, quel que soit le résultat du premier. Cela peut poser problème dans certains cas : si l'un des tests fait intervenir une expression non définie (par exemple une division par zéro ou une racine carrée d'un nombre négatif), une erreur peut survenir même si le résultat final n'en avait pas besoin. À l'inverse, dans la fonction `f5`, on n'entre dans le bloc `else` que si le premier test n'est pas validé. Cette structure évite d'évaluer des expressions inutiles (ou potentiellement non définies).

En revanche, la fonction `f5bis` présente un avantage important lorsqu'on travaille avec le module `numpy` : elle est vectorisée. Cela signifie qu'elle peut s'appliquer directement à tous les éléments d'un tableau (`array`) NumPy, sans qu'il soit nécessaire d'écrire une boucle explicite. Cette approche rend le code plus concis et souvent beaucoup plus rapide dans les traitements numériques.

- Avec `def` :

```
def f1(x) :
    return x**2+5

def f2(x,y) :
    return x+y-2

def f2bis(t) :
    return t[0]+t[1]-2

def f3(x) :
    return (x**2,x**3)

def f4(x,y) :
    return (x+y,x-y)
```

```
def f5(x) :
    if x<10:
        return 3
    elif x<15:
        return -2*x
    else:
        return x**2

# def f5(x) :
#     if x<10:
#         return 3
#     else:
#         if x<15:
#             return -2*x
#         else:
#             return x**2
```

```
# TESTS
print(f1(2))
print(f2(2,3))
print(f2bis((2,3)))
print(f2bis([2,3]))
print(f3(2))
print(f4(2,3))
print(f5(2),f5(12),f5(22))
```

```
9
3
3
3
(4, 8)
(5, -1)
3 -24 484
```

Exercice 6.12 (Jours de la semaine)

Constituez une liste "semaine" contenant le nom des sept jours de la semaine. En utilisant une boucle, écrivez le jour suivi du message suivant :

- "`Au boulot :(`" s'il s'agit du lundi au jeudi ;
- "`C'est le weekend :)`" s'il s'agit du vendredi ;
- "`C'est la fête!`" s'il s'agit du samedi ou du dimanche.

Input de la fonction : le jour (`str`) ; output de la fonction : le message (`str`) ; test : une boucle sur la liste `semaine`.

Correction

```
def afficher_message_jour(jour):
    # str -> str
    if jour in ["Lundi", "Mardi", "Mercredi", "Jeudi"]:
        return f"{jour} - Au boulot :("
    elif jour == "Vendredi":
        return f"{jour} - C'est le weekend :)"
    else:
        return f"{jour} - C'est la fête !"

# TEST
semaine = ["Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche"]
for j in semaine:
    resultat = afficher_message_jour(j)
    print(resultat)
```

```
Lundi - Au boulot :(                Jeudi - Au boulot :(                Dimanche - C'est la fête !
Mardi - Au boulot :(                Vendredi - C'est le weekend :)
Mercredi - Au boulot :(             Samedi - C'est la fête !
```

Exercice 6.13 (Pièces magiques - ter)

Dans l'exercice 4.6 on a construit une liste avec le montant des pièces cumulées chaque semaine. Maintenant on ne veut pas créer une liste mais simplement écrire une fonction qui renvoie le montant cumulée si on lui indique combien de semaines sont passées.

Correction

Input de type `int` ; output de type `int`.

```
pieces_cumulees = lambda semaine : 20 + 67 * semaine

for sem in [0, 1, 52]:
    print(f"À la fin de la semaine {sem}  on a {pieces_cumulees(sem)} pièces en tout.")
```

```
À la fin de la semaine 0  on a 20 pièces en tout.
À la fin de la semaine 1  on a 87 pièces en tout.
À la fin de la semaine 52  on a 3504 pièces en tout.
```

Exercice 6.14 (Le problème de la ferme)

Un fermier souhaite connaître le nombre total de pattes parmi tous ses animaux. Il élève trois types d'animaux : des poulets (2 pattes), des vaches (4 pattes) et des cochons (4 pattes). Le fermier vous donne un nombre d'animaux pour chaque espèce. Implémentez une fonction de signature `animals(poulets, vaches, cochons)` qui renvoie le nombre total de pattes pour tous les animaux.

Exemples : `animals(2, 3, 5)` renvoie 36 ; `animals(1, 2, 3)` renvoie 22 ; `animals(5, 2, 8)` renvoie 50.

Correction

Entrée : trois entiers (type `int`) ; sortie : un entier correspondant au nombre total de pattes.

```
animals = lambda poulets, vaches, cochons: (poulets * 2) + (vaches * 4) + (cochons * 4)

# TESTS aléatoires
```

```
from random import randint
for _ in range(5):
    poulets, vaches, cochons = randint(0, 10), randint(0, 10), randint(0, 10)
    total_pattes = animals(poulets, vaches, cochons)
    print(f"Poulets : {poulets:2d}, Vaches : {vaches:2d}, Cochons : {cochons:2d} ⇝ Pattes :
     ↪ {total_pattes:3d}")
```

```
Poulets :  7, Vaches :  8, Cochons :  5 ⇝ Pattes :  66
Poulets :  5, Vaches :  6, Cochons :  3 ⇝ Pattes :  46
Poulets :  4, Vaches :  1, Cochons :  1 ⇝ Pattes :  16
Poulets : 10, Vaches :  1, Cochons :  0 ⇝ Pattes :  24
Poulets :  0, Vaches :  2, Cochons :  9 ⇝ Pattes :  44
```

Le formatage avec `{variable:2d}` aligne les résultats pour une meilleure lisibilité. Les chiffres sont espacés de manière à garantir que les valeurs de poulets, vaches, et cochons soient affichées avec au moins 2 caractères de largeur, tandis que le total des pattes est aligné sur 3 caractères.

Exercice 6.15 (Poids sur la Lune)

À l'exercice 4.8 on a créé une boucle `for` pour afficher notre poids sur la Lune sur une période de 15 ans. Transformer cette boucle en une fonction.

Correction

Input : un `float` ; output : un `float`.

```
def poids_Terre_to_Lune(poids_sur_terre):
    facteur_lune = 0.165 # Facteur pour le poids sur la Lune
    poids_sur_lune = poids_sur_terre * facteur_lune
    return poids_sur_lune

poids_sur_terre = 70 # Poids sur Terre en kilogrammes
poids_sur_lune = poids_Terre_to_Lune(poids_sur_terre)
print(f"Poids sur Terre {poids_sur_terre} kg ⇝ poids sur Lune {poids_sur_lune:.2f} kg")
```

```
Poids sur Terre 70 kg ⇝ poids sur Lune 11.55 kg
```

Exercice 6.16 (Prix d'un billet)

Voici la réduction pour le prix d'un billet de train en fonction de l'âge du voyageur :

- réduction de 50% pour les moins de 10 ans ;
- réduction de 30% pour les 10 à 18 ans ;
- réduction de 20% pour les 60 ans et plus.

Écris une fonction qui renvoie la réduction en fonction de l'âge et dont les propriétés sont rappelées ci-dessous :

- Nom : `reduction()`
- Usage : `reduction(age)`
- Entrée : un entier correspondant à l'âge
- Sortie : un entier correspondant à la réduction
- Exemples : `reduction(17)` renvoie 30 ; `reduction(23)` renvoie 0

Écris une fonction qui calcule le montant à payer en fonction du tarif normal et de l'âge du voyageur :

- Nom : `montant()`
- Usage : `montant(tarif_normal,age)`
- Entrée : un nombre `tarif_normal` correspondant au prix sans réduction et `age` (un entier)
- Sortie : un nombre correspondant au montant à payer après réduction
- Remarque : utilise la fonction `reduction()`
- Exemples : `montant(100,17)` renvoie 70.

Considérons une famille qui achète des billets pour différents trajets, voici le tarif normal de chaque trajet et les âges des voyageurs :

- tarif normal 30 euros, enfant de 9 ans ;
- tarif normal 20 euros, pour chacun des jumeaux de 16 ans ;
- tarif normal 35 euros, pour chacun des parents de 40 ans.

Quel est le montant total payé par la famille ?

Correction
Rappel : une réduction de x% d'un prix p signifie payer $p \times \frac{100-x}{100}$.

```
def reduction(age):
    if age<=10: return 50
    elif age<=18: return 30
    elif age>=60: return 20
    else: return 0

def montant(tarif_normal,age):
    coeff=100-reduction(age)
    return tarif_normal*coeff/100

# TEST
Total = montant(30,9) + 2*montant(20,16) + 2*montant(35,40)
print(f'Montant total payé par la famille : {Total} euros')
```

```
Montant total payé par la famille : 113.0 euros
```

En version compacte :

```
#reduction = lambda age : 50 if (age<=10) else ( 30 if (age<=18) else ( 20 if (age>=60) else 0 ) )
reduction = lambda age : 50*(age<=10)+30*(10<age<=18)+20*(age>=60)
montant   = lambda tarif_normal,age : tarif_normal*(100-reduction(age))/100
# TEST
Total = montant(30,9) + 2*montant(20,16) + 2*montant(35,40)
print(f'Montant total payé par la famille : {Total} euros')
```

```
Montant total payé par la famille : 113.0 euros
```

Exercice 6.17 (Swap)

Écrire une fonction `Swap(L,i,j)` qui échange les éléments d'indices `i` et `j` dans une liste `L`. **La liste est modifiée directement, sans qu'une nouvelle liste soit créée ni qu'une valeur soit renvoyée.** Par exemple, pour `L = [0,1,2,10,80]` et `i=1`, `j=4`, l'appel `Swap(L,i,j)` modifiera la liste `L` qui maintenant sera égale à `[0,80,2,10,1]`.

Correction
Puisque la liste est un objet mutable, la fonction modifie la liste directement sans retourner de valeur (`return` n'est pas nécessaire).

```
def Swap(L,i,j):
    L[i],L[j] = L[j],L[i]

# Exemple d'utilisation
L = [0, 1, 2, 10, 80]
print("Avant l'échange :", L)
Swap(L, 1, 4)  # Échange des éléments au positions 1 et 4
print("Après l'échange :", L)
```

```
Avant l'échange : [0, 1, 2, 10, 80]
Après l'échange : [0, 80, 2, 10, 1]
```

Exercice 6.18 (CodEx - Casseur de digicode : fonction de fonction)

On cherche dans cet exercice à cracker un digicode, *i.e.* à déterminer la combinaison d'un digicode. Celui-ci est représenté par une fonction `digicode` prenant quatre paramètres `a`, `b`, `c` et `d`, correspondant chacun à un entier compris entre 0 et 9 (inclus), et renvoyant `True` si la combinaison formée par ces quatre chiffres est valide, `False` sinon. De façon générale, un digicode peut avoir plusieurs solutions. On cherche à en déterminer une (il en existe toujours au moins une dans l'énoncé). Par exemple la fonction `digicode = lambda a, b, c, d: (a + b - c - d == 0)`. L'appel `digicode(1, 2, 3, 4)` renvoie `False` donc (1, 2, 3, 4) n'est donc pas une combinaison valide ; l'appel `digicode(4, 5, 4, 5)` renvoie `True` donc (4, 5, 4, 5) est une combinaison valide mais aussi `digicode(1, 2, 1, 2)` renvoie `True` donc (1, 2, 1, 2) est une autre combinaison valide.

Écrire la fonction `digicrack(digicode)` qui **prend en paramètre une fonction** `digicode` et renvoie un tuple de

quatre entiers entre 0 et 9 représentant une combinaison valide acceptée par cette fonction.

Par exemple, si la fonction `digicode` (inconnue) est `digicode = lambda a, b, c, d: (a + b - c - d == 0)`, alors `digicrack(digicode)` peut renvoyer `(4, 5, 4, 5)` ou encore `(1, 2, 1, 2)` car `digicode(1,2,1,2)` renvoie `True`. Si `digicode = lambda a, b, c, d: (156*a - 820*b + 1463*c - 465*d == 7607 == 0)`, alors `digicrack(digicode)` peut renvoyer `(3, 5, 8, 1)` car `digicode(3, 5, 8, 1)` renvoie `True`.

Source : https://codex.forge.apps.education.fr/exercices/digicrack/

Correction

```
def digicrack(digicode):
    for a in range(10):
        for b in range(10):
            for c in range(10):
                for d in range(10):
                    if digicode(a, b, c, d):
                        return (a, b, c, d)

# Tests de démonstration
print(digicrack(lambda a,b,c,d: a+b-c-d == 0))
print(digicrack(lambda a,b,c,d: 156*a - 820*b + 1463*c - 465*d == 7607))
```

```
(0, 0, 0, 0)
(1, 0, 7, 6)
```

Plus lisible avec `itertools` :

```
from itertools import product

def digicrack_itertools(digicode):
    for comb in product(range(10), repeat=4):
        if digicode(*comb):
            return comb

# Tests de démonstration
print(digicrack(lambda a,b,c,d: a+b-c-d == 0))
print(digicrack(lambda a,b,c,d: 156*a - 820*b + 1463*c - 465*d == 7607))
```

```
(0, 0, 0, 0)
(1, 0, 7, 6)
```

Ou encore, “random crack” :

```
from random import randint

def digicrack_random(digicode):
    while True:
        a, b, c, d = (randint(0,9) for _ in range(4))
        if digicode(a,b,c,d):
            return (a,b,c,d)

# Tests de démonstration
print(digicrack(lambda a,b,c,d: a+b-c-d == 0))
print(digicrack(lambda a,b,c,d: 156*a - 820*b + 1463*c - 465*d == 7607))
```

```
(0, 0, 0, 0)
(1, 0, 7, 6)
```

Exercice Bonus 6.19 (`map`)

Soit $n \in \mathbb{N}$ (par exemple, $n = 123$). Que fait la commande `list(map(int,str(n)))` ? Écrire un script qui donne le même résultat sans utiliser `map` mais en utilisant une liste de compréhension.

Correction

```
>>> n = 123
>>> list(map(int,str(n)))
[1, 2, 3]
>>> [int(x) for x in str(n)]
[1, 2, 3]
```

Explications :

- `str(n)` transforme n en une chaîne de caractères
- `map(f,s)` équivaut à `f(x) for x in s`
- `list(M)` transforme `M` en une liste :

6.7.2. Exercices de révision

Exercice 6.20 (Cul de Chouette - version simplifiée)

Dans la série "Kaamelott" d'Alexandre Astier, le "cul de chouette" est le jeu favori du tavernier et du chevalier Karadoc. Le but présumé de ce jeu est de jeter des dés en tentant d'atteindre un certain total par jet.

Écrire une fonction qui, pour une valeur donnée, renvoie toutes les solutions de 3 dés (dés classiques allant de 1 à 6) pouvant donner cette valeur. Attention, les solutions doivent être uniques. Si la solution (1, 2, 3) convient pour la valeur 6 alors la solution (2, 3, 1) ne peut plus convenir (les dés sont interchangeables).

Correction

Source : https://github.com/E-delweiss/Cul_de_Chouette?tab=readme-ov-file

```
cdc = lambda total: [ (d1,d2,d3) for d1 in range(1,7) for d2 in range(d1,7) for d3 in range(d2, 7) if
↪  d1+d2+d3==total]

# TEST
total_a_trouver = 10
resultats = cdc(total_a_trouver)
print(f"Solutions pour total = {total_a_trouver} : {resultats}")
```

```
Solutions pour total = 10 : [(1, 3, 6), (1, 4, 5), (2, 2, 6), (2, 3, 5), (2, 4, 4), (3, 3, 4)]
```

Exercice 6.21 (Nombres de Friedman)

Les *nombres de Friedman* sont des nombres qui peuvent être exprimés en utilisant tous leurs chiffres dans une expression mathématique. Par exemple : 347 est un nombre de Friedman, car il peut être écrit sous la forme $4+7^3=347$; 127 est également un nombre de Friedman, car il peut s'écrire sous la forme $2^7-1=127$.

Implémentez une fonction qui vérifie si les expressions mathématiques donnée sous forme de chaîne de caractères donnent un nombre de Friedman.

Pour évaluer une chaîne de caractère on utilisera la fonction `eval()`. Exemple d'utilisation :

```
>>> s = "4+7**3"
>>> eval(s)
347
```

Correction

```
def est_friedman(n, expression):
    chiffres_utilisés = sorted([c for c in expression if c.isdigit()])
    chiffres_nombre = sorted(str(n))
    print(f"Chiffres utilisés dans {expression} : {chiffres_utilisés}")
    print(f"Nombre obtenu : {n}")
    print(f"Chiffres du nombre obtenu : {chiffres_nombre}")
    return chiffres_utilisés == chiffres_nombre

# TESTS
expressions = [ "7 + 3**6", "(3 + 4)**3", "3**6 - 5", "4 + 7^3", "2^7 - 1" ]
for expr in expressions:
    val = eval(expr)
    #print(f"{expr} = {val}")
    if est_friedman(val, expr):
        print(f"{val} est un nombre de Friedman")
    else:
        print(f"{val} n'est pas un nombre de Friedman")
    print("==================================================")
```

```
Chiffres utilisés dans 7 + 3**6 : ['3', '6', '7']
Nombre obtenu : 736
Chiffres du nombre obtenu : ['3', '6', '7']
736 est un nombre de Friedman
==================================================
Chiffres utilisés dans (3 + 4)**3 : ['3', '3', '4']
Nombre obtenu : 343
Chiffres du nombre obtenu : ['3', '3', '4']
343 est un nombre de Friedman
==================================================
Chiffres utilisés dans 3**6 - 5 : ['3', '5', '6']
Nombre obtenu : 724
Chiffres du nombre obtenu : ['2', '4', '7']
724 n'est pas un nombre de Friedman
==================================================
Chiffres utilisés dans 4 + 7^3 : ['3', '4', '7']
Nombre obtenu : 8
Chiffres du nombre obtenu : ['8']
8 n'est pas un nombre de Friedman
==================================================
Chiffres utilisés dans 2^7 - 1 : ['1', '2', '7']
Nombre obtenu : 4
Chiffres du nombre obtenu : ['4']
4 n'est pas un nombre de Friedman
==================================================
```

Exercice 6.22 (CodEx - Vérification de multiples)

Soient a et b deux entiers positifs. On dit que a est un multiple de b s'il existe un entier k tel que $a = b \times k$. Écrire (et tester) une fonction `est_multiple` qui vérifie si un entier est un multiple d'un autre. La fonction prend deux paramètres : a, un entier strictement positif ; b, un entier strictement positif. Elle retourne `True` si a est un multiple de b, et `False` dans le cas contraire.

Remarque : Python propose une solution native pour répondre à cette question (`%` ou `divmod`). On propose de raisonner différemment. Pour ce faire, on gère immédiatement le cas $b = 0$ en observant que le seul multiple de $b = 0$ est $a = 0$. Ce cas particulier étant traité, on procédera comme suite : on crée une variable `multiple` prenant initialement la valeur 0 ; tant que `multiple` est strictement inférieur à a, on ajoute b à la valeur de `multiple` ; en fin de boucle on teste l'égalité de `multiple` et a (s'ils sont égaux, a est un multiple de b, sinon a n'est pas un multiple de b).

Source : https://codex.forge.apps.education.fr/exercices/multiple/

Correction

```python
# Solution 1 : méthode native
# def est_multiple(a, b):
#     return a % b == 0

# Solution 2 : sans utiliser %
def est_multiple(a, b):
    if b == 0:
        return a == 0
    multiple = 0
    while multiple < a:
        multiple += b
    return multiple == a

# TESTS
for a, b in [ (0 ,  0), (5,  0), (10, 2),
              (10, 10), (2, 10), ( 7, 3)]:
    print(f"{a} est un multiple de {b} ?
    ↪ {est_multiple(a, b)}")
```

```
0 est un multiple de 0 ? True
5 est un multiple de 0 ? False
10 est un multiple de 2 ? True
10 est un multiple de 10 ? True
2 est un multiple de 10 ? False
7 est un multiple de 3 ? False
```

Exercice 6.23 (CodEx - Occurrences d'un caractère dans un mot)

Au premier chapitre, nous avons vu la méthode `.count`, qui renvoie le nombre d'occurrences d'un caractère dans une chaîne de caractères. Dans cet exercice, nous écrivons une fonction `compte_occurrences` qui réalise cette tâche **sans utiliser** `.count`. La fonction prend deux paramètres : `cible`, un caractère (chaîne de longueur 1), et `mot`, une chaîne de caractères. Elle retourne le nombre d'occurrences de `cible` dans `mot`.

Source : https://codex.forge.apps.education.fr/exercices/cpt_occurrences/

Correction

```python
# version 1 (non autorisée)
# compte_occurrences = lambda cible, mot : mot.count(cible)

# version 2
#def compte_occurrences(cible, mot):
#    cpt=0
#    for c in mot:
#        if c==cible:
```

```
#            cpt+=1
#    return cpt

# version 3
compte_occurrences = lambda cible, mot : sum(c == cible for c in mot )


# TESTS
for cible, mot in [ ("o", "bonjour"),  ("a", "abracadabra"), ("i", "abracadabra") ]:
    print(f"Il y a {compte_occurrences(cible, mot)} occurrences de la lettre {cible} dans le mot
    ↪ {mot}.")
```

```
Il y a 2 occurrences de la lettre o dans le mot bonjour.
Il y a 5 occurrences de la lettre a dans le mot abracadabra.
Il y a 0 occurrences de la lettre i dans le mot abracadabra.
```

Exercice 6.24 (CodEx - Occurrences d'un élément dans un tableau)

Dans cet exercice, nous allons créer une fonction `repetitions` qui prend en paramètre un élément `cible` et un tableau `valeurs`. La fonction doit renvoyer le nombre d'occurrences de `cible` dans `valeurs`. Vous ne devez pas utiliser les méthodes Python `count` ou toute autre méthode intégrée qui effectue cette tâche automatiquement.

Source: https://codex.forge.apps.education.fr/exercices/repetitions/

Correction

```
# version 1 non autorisée
# repetitions = lambda cible, valeurs : valeurs.count(cible)

# version 2
def repetitions(cible, valeurs):
    cpt = 0
    for v in valeurs:
        if v == cible:
            cpt += 1
    return cpt

# version 3
# repetitions = lambda cible, valeurs : sum(v == cible for v in valeurs )

# TESTS
for cible, valeurs in [ (2,[1, 2, 3, 2, 4]),  (1,[1, 1, 1, 1]), ("a",["b", "a", "c", "a", "d"]), (12,[])
↪ ]:
    print(f"L'élément {cible} apparaît {repetitions(cible, valeurs)} fois dans le tableau {valeurs}.")
```

```
L'élément 2 apparaît 2 fois dans le tableau [1, 2, 3, 2, 4].
L'élément 1 apparaît 4 fois dans le tableau [1, 1, 1, 1].
L'élément a apparaît 2 fois dans le tableau ['b', 'a', 'c', 'a', 'd'].
L'élément 12 apparaît 0 fois dans le tableau [].
```

Exercice 6.25 (CodEx - Conversion en minuscules)

Au premier chapitre, nous avons vu la méthode `.lower`, qui convertit une chaîne de caractères en lettres minuscules. Dans cet exercice, nous écrivons une fonction `convertit_minuscules` qui réalise cette tâche **sans utiliser** `.lower`. La fonction prend un paramètre `phrase`, une chaîne de caractères contenant potentiellement des majuscules, des minuscules, des chiffres et des symboles (les lettres sans accents). Elle retourne une nouvelle chaîne équivalente à `phrase`, où toutes les lettres majuscules ont été remplacées par leurs équivalents minuscules.

Source: https://codex.forge.apps.education.fr/exercices/conversion_chaines/

Correction

```
def convertit_minuscules(phrase):
    resultat = ""
    for c in phrase:
```

```
        if 'A' <= c <= 'Z':  # Vérifie si c est une majuscule
            resultat += chr(ord(c) + 32)  # Convertit en minuscule
        else:
            resultat += c  # Conserve les autres caractères
    return resultat

# Version non autorisée
# convertit_minuscules = lambda phrase : phrase.lower()

# TESTS
for phrase in ["Bonjour Tout Le Monde!", "PYTHON 3 EST GENIAL", "Voici Une Phrase Avec Majuscules"]:
    print(f" '{phrase}' ⇝ '{convertit_minuscules(phrase)}' ")
```

```
 'Bonjour Tout Le Monde!' ⇝ 'bonjour tout le monde!'
 'PYTHON 3 EST GENIAL' ⇝ 'python 3 est genial'
 'Voici Une Phrase Avec Majuscules' ⇝ 'voici une phrase avec majuscules'
```

Exercice 6.26 (CodEx - La moyenne olympique)

Lors des compétitions olympiques, la note finale d'un athlète est calculée selon une méthode particulière : on élimine les deux notes les plus extrêmes (la plus basse et la plus haute) et on calcule la moyenne des notes restantes. Les juges donnent des notes entières entre 0 et 10 (inclus), et vous devez écrire une fonction `moyenne_olympique` qui calcule cette moyenne olympique. Votre mission : implémenter cette fonction **sans utiliser** les fonctions `max`, `min`, `sum`, `sort` ou `sorted`.

Source : https://codex.forge.apps.education.fr/exercices/moy_olympic/

Correction

```
def moyenne_olympique(notes):
    min_note = max_note = total = notes[0]
    for note in notes[1:]:
        total += note
        if note < min_note:
            min_note = note
        elif note > max_note:
            max_note = note
    return (total - min_note - max_note) / (len(notes) - 2)

# TESTS
for notes in [ [2, 0, 2, 10, 2] , [1, 1, 1, 1, 1, 1] , [5, 1, 4, 3, 2, 6] ] :
    print(notes, moyenne_olympique(notes))
```

```
[2, 0, 2, 10, 2] 2.0
[1, 1, 1, 1, 1, 1] 1.0
[5, 1, 4, 3, 2, 6] 3.5
```

Exercice 6.27 (CodEx : Différences entre deux tableaux)

Un laboratoire reçoit des résultats depuis deux sources différentes. Les résultats sont des entiers regroupés dans deux tableaux `source_1` et `source_2` de même longueur. Quand deux résultats concordent, on considère qu'il n'y aura pas à refaire l'expérience, on le codera par `False`, mais si les résultats sont différents, on considère qu'il faudra refaire cette expérience, on le codera par `True` .

Complétez la fonction `differences(source_1, source_2)` qui prend en paramètres deux tableaux d'entiers de même longueur et renvoie un tableau de booléens de même taille.

Exemples :

- `differences([1,2,3],[1,0,3])` renvoie `[False, True, False]`
- `differences([5,5,5],[5,5,5])` renvoie `[False, False, False]`

Source : https://codex.forge.apps.education.fr/exercices/liste_differences/

Correction

```
def differences(source_1, source_2):
    return [ s1[i] != s2[i] for i in range(len(source_1)) ]
```

```
    # return [ s1 != s2 for s1, s2 in zip(source_1, source_2) ]

# Tests
TESTS = [
    ([14, 87, 22, 5, 65], [14, 86, 27, 5, 65]),
    ([1,2,3],[1,0,3]),
    ([5,5,5],[5,5,5]),
    ([10,20,30],[10,21,30]),
    ([0,0,0],[1,0,0]),
    ([-54], [-54])
]

from random import randint
length = randint(3,7)
s1 = [randint(0,10) for _ in range(length)]
s2 = [randint(0,10) for _ in range(length)]
TESTS.append((s2,s2))

for s1, s2 in TESTS:
    print(f"differences({s1},{s2}) = {differences(s1,s2)}")

differences([14, 87, 22, 5, 65],[14, 86, 27, 5, 65]) = [False, True, True, False, False]
differences([1, 2, 3],[1, 0, 3]) = [False, True, False]
differences([5, 5, 5],[5, 5, 5]) = [False, False, False]
differences([10, 20, 30],[10, 21, 30]) = [False, True, False]
differences([0, 0, 0],[1, 0, 0]) = [True, False, False]
differences([-54],[-54]) = [False]
differences([2, 1, 9, 8, 5, 10, 1],[2, 1, 9, 8, 5, 10, 1]) = [False, False, False, False, False, False,
↪  False]
```

Exercice 6.28 (Vérifier si une matrice est antisymétrique)

En algèbre linéaire, une *matrice antisymétrique* (ou *skew-symmetric matrix*) est une matrice carrée $\mathbb{A}$ telle que sa transposée est l'opposée d'elle-même : $\mathbb{A}^{\mathrm{T}} = -\mathbb{A}$. Autrement dit, pour tous les indices i, j,

$$a_{ij} = -a_{ji}.$$

1. Écrivez une fonction `est_antisymetrique(M)` qui prend en argument une matrice carrée (sous forme de liste de listes) et renvoie `True` si la matrice est antisymétrique, `False` sinon.
2. Testez votre fonction sur les matrices suivantes :

$$\mathbb{A}_1 = \begin{pmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{pmatrix}, \quad \mathbb{A}_2 = \begin{pmatrix} 0 & 1 & 2 \\ -1 & 1 & 1 \\ -2 & -1 & 0 \end{pmatrix}, \quad \mathbb{A}_3 = \begin{pmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -3 & -1 & 0 \end{pmatrix}.$$

Source : https://py.checkio.org/mission/skew-symmetric-matrix/

Correction

Pour qu'une matrice soit antisymétrique, chaque élément a_{ij} doit vérifier $a_{ij} = -a_{ji}$. On peut donc comparer la matrice initiale à l'opposée de sa transposée.

```
def est_antisymetrique(M):
    n = len(M)
    transpose_neg = [[-M[j][i] for j in range(n)] for i in range(n)]
    return M == transpose_neg

A1 = [
    [0, 1, 2],
    [-1, 0, 1],
    [-2, -1, 0]
]
print(est_antisymetrique(A1))
```

```
# attendu : True

True
```

```
A2 = [
    [0, 1, 2],
    [-1, 1, 1],
    [-2, -1, 0]
]
print(est_antisymetrique(A2))
```

```
# attendu : False
```

```
False
```

```
A3 = [
    [0, 1, 2],
    [-1, 0, 1],
    [-3, -1, 0]
]
print(est_antisymetrique(A3))
# attendu : False
```

```
False
```

Exercice 6.29 ($u_{n+1} = f(n, u_n)$)

Soit $(u_n)_{n\in\mathbb{N}}$ la suite définie par

$$\begin{cases} u_1 = 0, \\ u_{n+1} = n + 1 + \frac{u_n}{2}. \end{cases}$$

Vérifier que $u_n = 2(n-1)$.

Correction

Pour utiliser les mêmes indices qu'en python et donc stocker dans une liste en position n la valeur u_n, calculons d'abord u_0 :

$$u_1 = 0 + 1 + \frac{u_0}{2} \quad \rightsquigarrow \quad u_0 = -2.$$

```
def suite(N):
    uu = [-2]
    for n in range(N+1):
        uu.append(n+1+uu[-1]/2)
    return uu

print(suite(10))
```

```
[-2, 0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0, 20.0]
```

Exercice 6.30 (CodEx : détermination d'un seuil, suite de type ①)

Soit $(u_n)_{n\in\mathbb{N}}$ la suite définie par

$$u_n = \frac{5n^2 + 13}{n^2 + 2}.$$

On peut montrer que cette suite tend vers 5 lorsque n tend vers $+\infty$. Cela signifie que

$$\forall \varepsilon > 0 \quad \exists n \in \mathbb{N} \quad : \quad \forall n \geq \mathrm{N} \quad |u_n - 5| \leq \varepsilon.$$

Écrivez une fonction `seuil` qui prend en paramètre le nombre `eps` et renvoie la valeur du plus petit entier n tel que l'on ait $|u_n - 5| \leq \varepsilon$.

Remarque : au chapitre 4 on a vu différents types de suite. Il s'agit d'une suite de type ①.

Correction

Source : https://codex.forge.apps.education.fr/exercices/seuil_explicite/

```
def seuil(eps):
    u = lambda n : (5 * n**2 + 13) / (n**2 + 2)
    n = 1
    while abs( u(n) - 5) > eps: n += 1
    return n

# TESTS
for eps in [ 1, 0.1, 0.01, 0.001 ]:
    print(f"Le plus petit n tel que |u_n-5|<={eps} est {seuil(eps)}.")
```

```
Le plus petit n tel que |u_n-5|<=1 est 1.
Le plus petit n tel que |u_n-5|<=0.1 est 6.
Le plus petit n tel que |u_n-5|<=0.01 est 18.
Le plus petit n tel que |u_n-5|<=0.001 est 55.
```

Exercice 6.31 (Recherche de seuil : plus petit diviseur)

Pour un entier $n \geq 2$, écrire une fonction `plus_petit_diviseur(n)` qui renvoie le plus petit diviseur $d \geq 2$ de n.

Correction

On rappelle que d divise n si et seulement si `n%d==0`. Utiliser une boucle `for d in range(2,n+1)` est une mauvaise idée car dès qu'on trouve un diviseur, on sait qu'il sera le plus petit et il est inutile de continuer dans la boucle. Avec une boucle `while`, on continue à chercher tant qu'on n'a pas trouvé de diviseur (dès qu'on l'a trouvé, la boucle s'arrête).

```python
def plus_petit_diviseur(n):
    d = 2
    while n%d!=0 and d<=n: d += 1
    return d

# Exemple d'utilisation
n = 49
resultat = plus_petit_diviseur(n)
print(f"Le plus petit diviseur propre de {n} est {resultat}")
```

```
Le plus petit diviseur propre de 49 est 7
```

Exercice 6.32 (CodEx : détermination d'un seuil, suite de type ②)

Soit $(u_n)_{n\in\mathbb{N}}$ la suite définie par

$$u_0 = 1, \quad u_{n+1} = \frac{3}{4}u_n + 7.$$

On peut montrer que cette suite converge vers 28 lorsque n tend vers $+\infty$. Cela signifie que

$$\forall \varepsilon > 0 \quad \exists n \in \mathbb{N} \quad : \quad \forall n \geq \mathrm{N} \quad |u_n - 28| \leq \varepsilon.$$

Écrivez une fonction `seuil` qui prend en paramètre le nombre `eps` et renvoie la valeur du plus petit entier n tel que l'on ait $|u_n - 28| \leq \varepsilon$.

Remarque : au chapitre 4, nous avons vu différents types de suites. Il s'agit ici d'une suite récurrente de type ②.

Correction

Source : https://codex.forge.apps.education.fr/exercices/seuil_recurrence/

```python
def seuil(eps):
    f = lambda u : (3/4) * u + 7
    n = 0
    u = 1  # u_0
    while abs(u - 28) > eps:
        u = f(u)
        n += 1
    return n

# TESTS
for eps in [1, 0.1, 0.01, 0.001]:
    print(f"Le plus petit n tel que |u_n-28| <= {eps} est {seuil(eps)}.")
```

```
Le plus petit n tel que |u_n-28| <= 1 est 12.
Le plus petit n tel que |u_n-28| <= 0.1 est 20.
Le plus petit n tel que |u_n-28| <= 0.01 est 28.
Le plus petit n tel que |u_n-28| <= 0.001 est 36.
```

Exercice 6.33 (Détermination d'un seuil, suite de type ②)

Soit $(s_n)_{n\in\mathbb{N}}$ la suite définie par

$$s_n = \sum_{k=1}^{n} u_k, \qquad u_k = \frac{k^3}{2^k}.$$

Cette somme converge vers une limite ℓ finie et elle peut être écrite sous forme d'une suite définie par récurrence car

$$s_{n+1} = \sum_{k=1}^{n+1} u_k = \left(\sum_{k=1}^{n} u_k\right) + u_{n+1} = s_n + u_{n+1}$$

ainsi

$$\begin{cases} s_1 = f(1), \\ s_{n+1} = s_n + f(n+1), \end{cases} \qquad \text{avec} \qquad f\colon k \mapsto \frac{k^3}{2^k}.$$

En pratique, les nombres en machine ont une précision limitée. Cela signifie qu'au-delà d'un certain rang n, l'ajout d'un nouveau terme $f(n+1)$ devient si petit qu'il ne modifie plus la valeur de s_n stockée en mémoire. Autrement dit, on atteint un point où $s_{n+1} = s_n$ pour l'ordinateur. L'objectif est de déterminer le plus petit entier n tel que $s_n = s_{n+1}$ en machine et d'estimer la valeur de la limite atteinte.

Remarque : au chapitre 4, nous avons vu différents types de suites. Il s'agit ici d'une suite récurrente à un pas de type ②.

Correction

Avec stockage de tous les valeurs de la suite (s_n) :

```python
def seuil():

    u = lambda k: k**3 / 2**k

    n = 0
    ss = [u(n)]

    n = 1
    ss.append( ss[-1]+u(n) )

    while ss[-1] != ss[-2] :
        n += 1
        ss.append( ss[-1]+u(n) )

    return n, ss[-1]

# Execution
n_seuil, s_limite = seuil()
print(f"Le plus petit n tel que s_n devient constante en machine est {n_seuil}.")
print(f"Valeur de la limite approchée: {s_limite}")
```

```
Le plus petit n tel que s_n devient constante en machine est 68.
Valeur de la limite approchée: 26.0
```

Ce script conjecture que $s_n \to \ell = 26$. Comment le démontrer ? Soit

$$g\colon x \mapsto \sum_{n=0}^{\infty} x^n.$$

En dérivant g, on obtient

$$g'(x) = \sum_{n=0}^{\infty} n x^{n-1} = \sum_{n=1}^{\infty} n x^{n-1}.$$

On multiplie cette expression par x et on dérive à nouveau

$$x g'(x) = \sum_{n=1}^{\infty} n x^n \qquad \rightsquigarrow \qquad x g''(x) + g'(x) = \sum_{n=1}^{\infty} n^2 x^{n-1}.$$

Une nouvelle multiplication par x suivie d'une différentiation conduit à

$$x^2 g''(x) + x g'(x) = \sum_{n=1}^{\infty} n^2 x^n \qquad \rightsquigarrow \qquad x^2 g'''(x) + 3x g''(x) + g'(x) = \sum_{n=1}^{\infty} n^3 x^{n-1}.$$

Si on évalue le terme de droite en $\frac{1}{2}$ on trouve

$$\sum_{n=1}^{\infty} \frac{n^3}{2^{n-1}} = 2 \sum_{n=1}^{\infty} \frac{n^3}{2^n} = 2\ell.$$

D'autre part, on sait que, pour $|x| < 1$, $g(x) = \frac{x}{1-x}$ ainsi

$$g(x) = \frac{1}{1-x} - 1, \qquad g'(x) = \frac{1}{(1-x)^2}, \qquad g''(x) = \frac{2}{(1-x)^3}, \qquad g'''(x) = \frac{6}{(1-x)^4}$$

ainsi

$$\underbrace{\left(\frac{1}{2}\right)^2 g'''(1/2) + 3\left(\frac{1}{2}\right) g''(1/2) + g'(1/2)}_{\frac{1}{4}\times 96+\frac{3}{2}\times 16+4=24+24+4=52} = 2\ell.$$

Exercice 6.34 (Suites type ③ : récurrence à deux pas $u_{n+1} = f(u_n, u_{n-1})$)

Soit $(u_n)_{n\in\mathbb{N}}$ la suite définie par récurrence

$$\begin{cases} u_0 = 0, \\ u_1 = 1, \\ u_{n+1} = 3u_n + 2u_{n-1}. \end{cases}$$

Écrivez le code de la fonction `suite` qui prend en paramètre le nombre `seuil` et renvoie la valeur u_{N} et le rang N du premier terme de cette suite strictement supérieur à seuil.

Remarque : au chapitre 4, nous avons vu différents types de suites. Il s'agit ici d'une suite récurrente à deux pas de type ③.

Correction

On ne connaît pas combien d'itérations seront nécessaire, on utilisera donc une boucle `while`.

```python
def suite(seuil):
    f = lambda a, b: 3 * a + 2 * b
    u0, u1 = 0, 1
    # print(f"u_{0}={u0}")
    # print(f"u_{1}={u1}")
    n = 1
    while u1 <= seuil:
        u0, u1 = u1, f(u1, u0)  # Stockage de u_n dans u0 et de  u_{n+1} dans u1
        n += 1
        # print(f"u_{n}={u1}")
    return n, u1

# Test
n, u = suite(1000)
print(f"Rang: {n}, Valeur: {u}")
```

```
Rang: 7, Valeur: 1763
```

Exercice 6.35 (Moyenne Arithmétique-Géométrique et Constante de Gauss)

Soient deux réels positifs x et y. La moyenne arithmétique-géométrique de x et y est définie comme la limite des deux suites récurrentes :

$$\begin{cases} a_0 = x, \\ b_0 = y, \\ a_{n+1} = \dfrac{a_n + b_n}{2}, \\ b_{n+1} = \sqrt{a_n b_n}. \end{cases}$$

Ces suites convergent toutes deux vers un même nombre, que l'on appelle la moyenne arithmétique-géométrique de x et y, notée $agm(x, y)$.

Écrivez une fonction `agm(x, y, tol=1e-14)` qui calcule la moyenne arithmétique-géométrique de deux nombres réels x et y jusqu'à ce que la différence entre les deux suites définies ci-dessus soit inférieure à une tolérance donnée tol. Utilisez cette fonction pour calculer la constante de Gauss, qui est définie comme $\frac{1}{agm(1,\sqrt{2})}$. Affichez le résultat avec une précision de 10^{-14}.

Remarque : au chapitre 4, nous avons vu différents types de suites. Il s'agit ici d'une suite récurrente de type ④ :

$(u, v)_{n+1} = (f_1(u_n, v_n), f_2(u_n, v_n))$.

Correction
Source : https://scipython.com/book/chapter-2-the-core-python-language-i/questions/the-arithmetic-geometric-mean/

```python
def agm(x, y, tol=1e-14):
    a_n, b_n = x, y
    while abs(a_n - b_n) > tol:
        a_n, b_n = (a_n + b_n) / 2, (a_n*b_n)**0.5
    return a_n, b_n

# TEST
tol = 1.e-14
x, y = 1, 2**0.5
a, b = agm(x, y)
print(f'La constante de Gauss est : G = {1/a:.14f} = {1/b:.14f}')
```

```
La constante de Gauss est : G = 0.83462684167407 = 0.83462684167407
```

Exercice 6.36 (Schémas d'analyse numérique : recherche de zéros)

On considère une fonction continue $f : [a, b] \to \mathbb{R}$. Le but de cet exercice est de *calculer numériquement* une solution de l'équation $f(x) = 0$, c'est-à-dire d'approcher un réel ℓ tel que $f(\ell) = 0$. On utilise pour cela différents *schémas d'analyse numérique*, qui produisent (sous certaines hypothèses) des suites convergeant vers ℓ.

Schéma 1. **Dichotomie (ou bissection).** On suppose que l'on connaît un intervalle initial $[a_0, b_0]$ tel que $f(a_0)$ et $f(b_0)$ soient de signes opposés : $f(a_0)f(b_0) < 0$, ce qui garantit l'existence d'au moins un zéro de f dans cet intervalle. À chaque étape, on définit le milieu de l'intervalle :

$$x_n = \frac{a_n + b_n}{2}.$$

On choisit ensuite le sous-intervalle $[a_{n+1}, b_{n+1}]$ dans lequel f change de signe, c'est-à-dire $[a_{n+1}, b_{n+1}] = [a_n, x_n]$ si $f(a_n)f(x_n) < 0$, sinon $[a_{n+1}, b_{n+1}] = [x_n, b_n]$. Les intervalles obtenus sont de plus en plus petits et contiennent tous une solution.

Schéma 2. **Méthode de Lagrange (fausse position).** Comme pour la dichotomie, on suppose qu'il existe $a_0 < b_0$ tels que $f(a_0)f(b_0) < 0$. On remplace le milieu de l'intervalle par l'intersection de l'axe des abscisses avec la droite reliant les points $(a_n, f(a_n))$ et $(b_n, f(b_n))$:

$$x_{n+1} = b_n - \frac{f(b_n)(b_n - a_n)}{f(b_n) - f(a_n)}.$$

On conserve ensuite un sous-intervalle $[a_{n+1}, b_{n+1}]$ contenant un changement de signe comme dans le cas de la dichotomie.

Schéma 3. **Méthode de la sécante.** On construit directement une suite (x_n) à partir de deux valeurs initiales x_0 et x_1 :

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

Géométriquement, x_{n+1} est l'abscisse du point d'intersection de la droite qui relie $(x_n, f(x_n))$ et $(x_{n-1}, f(x_{n-1}))$ avec l'axe des abscisses.

Schéma 4. **Méthode de Newton (avec f dérivable).** On suppose ici que f est dérivable et que l'on connaît sa dérivée f'. À partir d'une valeur initiale x_0, on définit la suite

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Géométriquement, x_{n+1} est l'abscisse du point d'intersection de la tangente à la courbe de f en x_n avec l'axe des abscisses.

Travail demandé.

1. Écrire une fonction Python pour chacune de ces méthodes.

2. Chaque fonction prendra en paramètres la fonction f (et éventuellement sa dérivée f'), les valeurs initiales nécessaires, ainsi qu'un seuil $\varepsilon > 0$.
3. La fonction devra renvoyer une approximation numérique du zéro de f et le nombre d'itérations effectuées.
4. Tester les programmes sur une fonction simple admettant un zéro connu et comparer la rapidité de convergence des différentes méthodes.

Correction

```python
def dichotomie(f, a, b, eps):
    n = 0
    while b - a > eps:
        m = (a + b) / 2
        if f(a)*f(m) <= 0: b = m
        else: a = m
        n += 1
    return (a + b) / 2, n
```

```python
def secante(f, x0, x1, eps):
    n = 0
    while abs(f(x1)) > eps:
        f0, f1 = f(x0), f(x1)
        x0, x1 = x1, x1-f1*(x1-x0)/(f1-f0)
        n += 1
    return x1, n
```

```python
def lagrange(f, a, b, eps):
    n = 0
    x = b
    while abs(f(x)) > eps:
        fa, fb = f(a), f(b)
        x = b - fb*(b-a)/(fb-fa)
        if f(a)*f(x) <= 0: b = x
        else: a = x
        n += 1
    return x, n
```

```python
def newton(f, df, x0, eps):
    n = 0
    while abs(f(x0)) > eps:
        x0 = x0 - f(x0)/df(x0)
        n += 1
    return x0, n
```

```python
# Exemple de test
f  = lambda x: x**2 - 5
df = lambda x: 2*x

print("Dichotomie :", dichotomie(f, 2, 3, 1e-6))
print("Lagrange   :", lagrange(f, 2, 3, 1e-6))
print("Sécante    :", secante(f, 2, 3, 1e-6))
print("Newton     :", newton(f, df, 3, 1e-6))
```

```
Dichotomie : (2.2360682487487793, 20)
Lagrange   : (2.236067926333413, 8)
Sécante    : (2.236067926333413, 4)
Newton     : (2.236067977499978, 4)
```

Exercice 6.37 (Aucune Condition)

Écrivez une fonction qui renvoie 0 si l'entrée est 1, et renvoie 1 si l'entrée est 0. Essayez de relever ce défi sans utiliser ni de `if`, ni d'opérateurs ternaires, ni de négations.

Correction

L'idée est de tirer parti de la propriété mathématique que $1-0=1$ et $1-1=0$, qui correspond à ce que nous voulons obtenir dans la fonction `flip`. Cela nous permet d'éviter l'utilisation de déclarations conditionnelles (`if-else`) ou d'autres opérateurs spéciaux.

```python
flip = lambda x: 1 - x
# TESTS
for num in [0,1]:
    print(f"{num} ⇝ {flip(num)}")
```

```
0 ⇝ 1
1 ⇝ 0
```

Exercice 6.38 (Recréer la Fonction `abs()`)

La fonction `abs()` retourne la valeur absolue d'un nombre. Vous pouvez la considérer comme la distance par rapport à zéro. Implémenter une fonction `myabs` qui recrée cette fonctionnalité **sans** utiliser la fonction prédéfinie.

Bonus : et sans utiliser de conditions ?

Correction

Voici quelques implémentations possibles :

①
```
def myabs(x):
    if x<0:
        x=-x
    return x
# TESTS
print(myabs(-1),myabs(0),myabs(1))
```
```
1 0 1
```

② Avec un opérateur ternaire :
```
myabs = lambda x : x if (x>=0) else -x
# TESTS
print(myabs(-1),myabs(0),myabs(1))
```
```
1 0 1
```

③ On utilise la propriété `a*True=a` et `a*False=0` :
```
myabs = lambda x : (x>=0)*(x) + (x<0)*(-x)
# TESTS
print(myabs(-1),myabs(0),myabs(1))
```
```
1 0 1
```

④ On utilise la propriété $|a-b| = \max(a,b) - \min(a,b)$:
```
myabs = lambda x : max(x,0)-min(x,0)
# TESTS
print(myabs(-1),myabs(0),myabs(1))
```
```
1 0 1
```

Exercice 6.39 (Vérification de Doublons)

Écrire une fonction qui prend en entrée une liste (ou un tuple) L et renvoie le booléen `True` si elle contient au moins deux éléments identiques, `False` sinon.

Correction

`set(L)` crée un ensemble à partir de la liste L. Un ensemble ne peut pas contenir de doublons, donc s'il y a des doublons dans la liste, la longueur de l'ensemble sera inférieure à la longueur de la liste. La condition `len(L) == len(set(L))` vérifie si la longueur de la liste est égale à la longueur de l'ensemble, ce qui signifie qu'il n'y a pas de doublons.

```
doublons = lambda L : len(L) > len(set(L))

print(doublons(['a','b','a']), doublons([1,2,5,89]))
```
```
True False
```

Exercice 6.40 (Entier manquant)

Implémenter une fonction `manque` qui, pour une liste de N − 1 entiers distincts tous compris entre 1 et N, renvoie l'entier manquant.

Correction

Notons m le numéro manquant. On sait que $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$. Dans notre cas N =`len(L)`+1 et `sum(L)`$+m = \frac{N(N+1)}{2}$.

```
manque = lambda L: (len(L)+2)*(len(L)+1)//2 - sum(L)

# TEST random
from random import randint
# Générer une liste de longueur N aléatoire
N = randint(10, 20)
L = list(range(1, N+1))
# Index aléatoire à retirer
idx = randint(0, N-1)
L.pop(idx)
print(f"{L = }, {manque(L) = }")
```
```
L = [1, 2, 3, 4, 5, 6, 7, 8, 9], manque(L) = 10
```

Exercice 6.41 (Pair ou Impair)

Implémenter une fonction `pairImpair` qui, pour un nombre entier N donné, renvoie la chaîne de caractères "pair" si N est pair, et "impair" si N est impair.

Correction

Entrée : un entier (`int`). Sortie : une chaîne de caractères (`str`) indiquant si l'entier est pair ou impair.

① Solution 1 : utilisation d'une condition `if`

```
def pairImpair(N):
    # int -> str
    if N%2 == 0:
        return "pair"
    else:
        return "impair"

# lambda N: "pair" if N%2 == 0 else "impair"

print(pairImpair(2), pairImpair(3))
pair impair
```

② Solution 2 : utilisation d'une liste

```
def pairImpair(N):
    L = ["pair", "impair"]
    return L[N%2]

# lambda N: ["pair", "impair"][N%2]

print(pairImpair(2), pairImpair(3))
pair impair
```

Exercice 6.42 (Divisibilité)
Écrire une fonction `mul2014` qui renvoie 1 si n est un multiple de 2014, 0 sinon.

Correction
Input de type `int`, output de type `int`.

- La version 1 utilise une structure classique `if...else` pour vérifier si le nombre est un multiple de 2014 et renvoie 1 dans ce cas, sinon 0.
- La version 2 utilise une syntaxe plus concise en utilisant une expression ternaire "`1 if n%2014 == 0 else 0`".
- La version 3 définit une fonction anonyme (lambda) avec la même logique que la version 2.
- La version 4 utilise la propriété selon laquelle $1 \times \text{True} = 1$ et $1 \times \text{False} = 0$, en multipliant l'expression booléenne `n%2014==0` par 1.
- La version 5 utilise une technique similaire à la version 4 mais utilise une liste `[0,1]` et l'expression booléenne `n%2014==0` pour choisir l'indice correspondant à 0 ou 1 dans la liste.

```
def mul2014_v1(n):
    if n%2014 == 0 :
        return 1
    else :
        return 0

def mul2014_v2(n):
    return 1 if n%2014 == 0 else 0

print(mul2014_v1(2014), mul2014_v1(2015))
print(mul2014_v2(2014), mul2014_v2(2015))
```

```
1 0
1 0
```

```
mul2014_v3 = lambda n : 1 if n%2014==0 else 0
mul2014_v4 = lambda n : 1*(n%2014==0)
mul2014_v5 = lambda n : [0,1][n%2014==0]

print(mul2014_v3(2014), mul2014_v3(2015))
print(mul2014_v4(2014), mul2014_v4(2015))
print(mul2014_v5(2014), mul2014_v5(2015))
```

```
1 0
1 0
1 0
```

Exercice 6.43 (Liste impairs)
Écrire une fonction lambda qui prend en entrée deux entiers L, R (avec $L < R$) et renvoie la liste des nombre impairs entre L et R (inclus).

Correction
On peut d'abord construire la liste des impairs avec une compréhension de liste

```
impaires = [i for i in range(L,R+1) if i%2!=0]    # ou i%2 == 1
```

Transformons ce raisonnement en une fonction lambda : *input* deux objets de type `int`, *output* un objet de type `list` de `int`, implémentation :

```
impaires = lambda L,R : [i for i in range(L,R+1) if i%2!=0]
print(impaires(1,15))
```

```
[1, 3, 5, 7, 9, 11, 13, 15]
```

Variante : utiliser directement `range` avec un pas de 2 en ajustant le point de départ (si L est pair, commencer à L + 1, sinon à L).

```
impaires = lambda L, R: [i for i in range(L + (L % 2 == 0), R+1, 2)]
print(impaires(1, 15))
```

Cette version évite de filtrer tous les nombres et ne génère que les impairs.

Exercice 6.44 (Liste divisibles)

Écrire une fonction lambda qui prend en entrée deux entiers L, R, d (on supposera $L < R$) et renvoie la liste des nombre entre L et R (inclus) qui sont divisibles par d.

Correction

```
>>> divisibles = lambda L,R,d : [i for i in range(L,R+1) if i%d==0]
>>> print(divisibles(50,100,7))
[56, 63, 70, 77, 84, 91, 98]
```

Exercice 6.45 (Nombres parfaits)

Pour un entier naturel $n \in \mathbb{N}$ donné, on définit $d(n)$ comme étant la somme de ses **diviseurs propres** (c'est-à-dire tous les diviseurs de n sauf n lui-même).

- Si $d(n) = n$ on dit que n est parfait,
- si $d(n) < n$ on dit que n est déficient,
- si $d(n) > n$ on dit que n est abondant.

Par exemple,

$$\left.\begin{aligned} a &= 220 \text{ est divisible par } 1,2,4,5,10,11,20,22,44,55,110 \\ d(a) &= 1+2+4+5+10+11+20+22+44+55+110 = 284 > a \end{aligned}\right] \Longrightarrow \quad a \text{ est abondant}$$

$$\left.\begin{aligned} b &= 284 \text{ est divisible par } 1,2,4,71,142 \\ d(b) &= 1+2+4+71+142 = 220 < b \end{aligned}\right] \Longrightarrow \quad b \text{ est déficient}$$

$$\left.\begin{aligned} c &= 28 \text{ est divisible par } 1,2,4,7,14 \\ d(c) &= 1+2+4+7+14 = 28 = c \end{aligned}\right] \Longrightarrow \quad c \text{ est parfait}$$

Objectif : écrire une fonction qui, pour N donné, classe tous les nombres $1 \leq n \leq N$ en trois catégories : abondants, déficients et parfaits.

Correction

On utilise une boucle pour examiner les nombres de 1 à N inclus. Pour chaque nombre n, on identifie tous les diviseurs propres de n en utilisant une compréhension de liste. Ensuite, on calcule la somme de ces diviseurs et on les compare avec n pour déterminer la catégorie à laquelle n appartient. Les nombres abondants sont stockés dans la liste A, les nombres déficients dans D et les nombres parfaits dans P.

```
def classer(N):
    A, D, P = [], [], []  # Listes pour Abondants, Déficients, Parfaits
    for n in range(1, N + 1):
        diviseurs = [x for x in range(1, int(n / 2) + 1) if (n%x==0)]
        s = sum(diviseurs)  # Somme des diviseurs
        # Phase de debug : afficher n, s et la liste des diviseurs
        # print(f"n={n:3d}, s={s:3d}, diviseurs={diviseurs}")
        if s == n:    P.append(n)
        elif s < n:   D.append(n)
        else:         A.append(n)
    return P, A, D

# Appel de la fonction avec N=30
P, A, D = classer(30)
# Affichage des résultats
print("Abondants:", A)
print("Déficients:", D)
print("Parfaits:", P)
```

```
Abondants: [12, 18, 20, 24, 30]
Déficients: [1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 19, 21, 22, 23, 25, 26, 27, 29]
Parfaits: [6, 28]
```

Exercice 6.46 (Radars routiers)

Les radars routiers servent à mesurer la vitesse des véhicules et à les sanctionner s'ils dépassent la limite autorisée. Comme pour toute mesure, des incertitudes subsistent. Ainsi, la loi prévoit qu'en cas de mesure par un radar fixe, une certaine marge d'erreur est appliquée. Si la vitesse mesurée est inférieure à 100 km/h, 5 km/h sont retranchés. Si la vitesse mesurée dépasse 100 km/h, 5% de la vitesse sont soustraits. Le résultat final représente la vitesse retenue pour le véhicule, déterminant ainsi si elle dépasse la limite de vitesse. Par exemple, pour une vitesse mesurée de 60 km/h, la vitesse retenue serait 55 km/h, et pour une vitesse mesurée de 110 km/h, la vitesse retenue serait 104.5 km/h.

Écrire une fonction qui prend en entrée la vitesse mesurée v et renvoie la vitesse retenue pour la verbalisation après le retrait de la marge d'erreur spécifiée par la loi.

Correction

```python
# Version 1
#def v_retenue(v):
#    if v<100:
#        return v-5
#    else:
#        return v*0.95

# Version 2
v_retenue = lambda v : (v-5) if (v<100) else (v*0.95)

# Version 3
# v_retenue = lambda v : (v-5)*(v<100)+(v*0.95)*(v>=100)

# TESTS
for v in [60,100,110]:
    print(f"vitesse = {v}, vitesse retenue = {v_retenue(v)}")
```

```
vitesse = 60, vitesse retenue = 55
vitesse = 100, vitesse retenue = 95.0
vitesse = 110, vitesse retenue = 104.5
```

Exercice 6.47 (Multiple de 10 le plus proche)

Soit $n \in \mathbb{N}^*$. Écrire une fonction qui arrondi n au multiple de 10 le plus proche. Si deux multiples sont à la même distance, elle renvoie le plus petit.

Correction

Input de type `int`, output de type `int`.

Voici deux approches pour résoudre le problème : la première utilise la division entière et le modulo pour effectuer les calculs, tandis que la deuxième convertit d'abord le nombre en chaîne de caractères pour manipuler les chiffres individuels.

La première méthode utilise la division entière pour obtenir le quotient et le reste de la division de N par 10. Ensuite, elle vérifie si le reste est inférieur ou égal à 5 afin d'arrondir au multiple de 10 le plus proche. Si c'est le cas, elle retourne le quotient multiplié par 10. Sinon, elle retourne le quotient multiplié par 10 et additionné de 10.

```
def roundToNearest(N):
    q,r = divmod(N,10)
    if r<=5:
        return q*10
    else:
        return q*10+10

print(f" {roundToNearest(15) = }")
print(f" {roundToNearest(29) = }")
```

```
 roundToNearest(15) = 10
 roundToNearest(29) = 30
```

Quant à la deuxième fonction, elle commence par convertir N en une chaîne de caractères pour travailler avec les chiffres individuels. Elle récupère les dizaines en remplaçant le dernier chiffre par un zéro. Ensuite, elle vérifie si le dernier chiffre de N est inférieur ou égal à 5 pour décider de retourner la valeur des dizaines ou la valeur des dizaines plus 10.

```
def roundToNearest(N):
    s = str(N)
    d = int(s[:-1]+"0")
    if int(s[-1])<=5:
        return d
    else:
        return d+10

print(f" {roundToNearest(15) = }")
print(f" {roundToNearest(29) = }")
```

```
 roundToNearest(15) = 10
 roundToNearest(29) = 30
```

Une autre implémentation de la première méthode est la suivante (sans calculer le quotient) :

```
def roundToNearest(N):
    r = N%10
    if r<=5:
        return N-r
    else:
        return N-r+10

# IDEM
# roundToNearest = lambda N : N-(N%10) if (N%10)<=5 else N-(N%10)+10
# ameliorée par l'utilisation de l'opérateur morse pour capturer et réutiliser le reste du modulo
# roundToNearest = lambda N : N-r if (r:=N%10)<=5 else N-r+10
```

On pourrait être tenté d'utiliser tout simplement la fonction `round(N)` qui arrondit N à l'entier le plus proche :

```
roundToNearest = lambda N: round(N/10)*10
```

Problème : pour les nombres exactement à mi-chemin entre deux entiers (par exemple, 0.5), elle arrondit au nombre PAIR le plus proche. Par exemple, `round(2.5)` est égal à 2, tandis que `round(3.5)` est égal à 4. Cela est dû à une convention d'arrondi appelée "arrondi par la parité".

Exercice 6.48 (Rugby)

Si on connaît le score d'un match de rugby, peut-on trouver toutes les façons de l'obtenir ? Les règles simplifiées du rugby à XV sont les suivantes : un essai rapporte 5 points ; après chaque essai, on peut tenter une transformation qui rapporte 2 points supplémentaires ; un coup de pied (pénalité ou drop) rapporte 3 points.

Créer une fonction `generer_combinaisons(score_cible)` qui génère toutes les façons possibles de marquer un certain nombre de points pour une équipe. Chaque combinaison doit être représentée par une liste de trois nombres entiers : nombre d'essais, nombre de transformations (doit être inférieur ou égal au nombre d'essais), nombre de coups de pied (pénalités ou drops).

Par exemple, pour un score de 24 points, la fonction pourrait renvoyer des combinaisons comme `[4,2,0]` car $4\times5+2\times2+0\times3=24$ ou encore `[3,3,1]` car $3\times5+3\times2+1\times3=24$, etc.

Correction

https://emilypython.wordpress.com/2019/06/16/les-scores-au-rugby/

```
def generer_combinaisons(score_cible):
    max_essais = score_cible // 5
    max_cdp = score_cible // 3
    combinaisons = [
        [essais, transfos, cdp]
        for essais in range(max_essais+1)
```

```
        for transfos in range(essais+1)  # transformation ≤ essais
        for cdp in range(max_cdp+1)
        if essais * 5 + transfos * 2 + cdp * 3 == score_cible
    ]
    return combinaisons


# TESTS
pts = 24
resultats = generer_combinaisons(pts)
print(f"Il y a {len(resultats)} façons possibles de marquer {pts} points :")
print(*resultats,sep="\t")
```

```
Il y a 5 façons possibles de marquer 24 points :
[0, 0, 8]→[2, 1, 4]→[3, 0, 3]→[3, 3, 1]→[4, 2, 0]
```

Exercice Bonus 6.49 (Homer et la Duff)

Après une longue journée à la centrale nucléaire, Homer Simpson se rend à son bar préféré, le Moe's Tavern, et commence à boire à une certaine heure T (en format 24 heures). Il consomme N pintes de sa bière préférée, la Duff. Chaque pinte de Duff contient 3 unités d'alcool standard (bière forte - environ 5.2%). Le corps d'Homer est capable de métaboliser 1 unité d'alcool par heure.

En France, la limite légale pour conduire est de 0.5 g/L d'alcool dans le sang, ce qui correspond environ à 1 unité d'alcool. L'objectif est de déterminer à quelle heure Homer pourra légalement conduire à nouveau, c'est-à-dire lorsque son taux d'alcool sera retombé à 1 unité ou moins. Il faut calculer l'heure exacte H et le nombre de jours D qu'Homer devra attendre avant de pouvoir conduire. Si l'heure dépasse 24 heures, cela signifie qu'il pourra conduire le jour suivant.

Par exemple, considérons le cas où Homer boit 1 pinte de bière ($N = 1$) à 19:00 ($T = 19$).

- En buvant 1 pinte de bière, Homer aura dans le sang $1 \times 3 = 3$ unités d'alcool.
- Son corps métabolise 1 unité d'alcool par heure.
- Après 2 heures, il aura métabolisé 2 unités, et il lui restera $3 - 2 = 1$ unité d'alcool dans le sang, ce qui est le seuil légal pour pouvoir conduire.
- Il pourra donc conduire à nouveau à 21:00 ($H = 21$) du même jour ($D = 0$).

Correction

```
def heure_legale_pour_conduire(N, T):
    total_unites = N * 3
    unites_a_metaboliser = total_unites - 1
    heures_necessaires = max(unites_a_metaboliser, 0)
    H = T + heures_necessaires
    D, H = divmod(H,24)
    return H, D

# TESTS
N = 1  # Nombre de pintes de bière
T = 19  # Heure de début (19h00)
H, D = heure_legale_pour_conduire(N, T)
print(f"Avec {N} pintes de bière à {T} heures, Homer pourra conduire dans {D} jour(s) à {H}:00.")

N, T = 5, 17
H, D = heure_legale_pour_conduire(N, T)
print(f"Avec {N} pintes de bière à {T} heures, Homer pourra conduire dans {D} jour(s) à {H}:00.")

N, T = 20, 23
H, D = heure_legale_pour_conduire(N, T)
print(f"Avec {N} pintes de bière à {T} heures, Homer pourra conduire dans {D} jour(s) à {H}:00.")
```

```
Avec 1 pintes de bière à 19 heures, Homer pourra conduire dans 0 jour(s) à 21:00.
Avec 5 pintes de bière à 17 heures, Homer pourra conduire dans 1 jour(s) à 7:00.
Avec 20 pintes de bière à 23 heures, Homer pourra conduire dans 3 jour(s) à 10:00.
```

Exercice 6.50 (Mélange à queue d'aronde)

Le `mélange à queue d'aronde` consiste à couper un jeu de carte en deux moitiés égales, puis de les intercaler parfaitement : une carte du tas de gauche, une carte du tas de droite, une carte du tas de gauche etc.

Soit `deck` une liste ayant un nombre pair d'éléments. Écrire une fonction `perfect_shuffle(deck)` qui renvoie une nouvelle liste construite selon ce mélange.

Par exemple, `perfect_shuffle([1, 2, 3, 4, 5, 6])=[1, 4, 2, 5, 3, 6]`.

Vérifier ensuite que, pour une liste de 1024 éléments, après 10 mélange on retrouve la disposition initiale.

Correction

```python
def perfect_shuffle(deck):
    n = len(deck)//2
    L = deck[:n]
    R = deck[n:]
    deck_NEW = []
    for i in range(n):
        deck_NEW.extend([L[i],R[i]])
    return deck_NEW

# def perfect_shuffle2(deck):
#     n = len(deck)//2
#     L = deck[:n]
#     R = deck[n:]
#     return [i for pair in zip(L, R) for i in pair]
```

```python
# TEST 1
deck_1 = list(range(1,7))
deck_2 = perfect_shuffle(deck_1)
print(deck_1)
print(deck_2)

# TEST 2
deck_1 = list(range(1,1025))
deck_2 = perfect_shuffle(deck_1)
for _ in range(9):
    deck_2 = perfect_shuffle(deck_2)
print(deck_1==deck_2)
```

```
[1, 2, 3, 4, 5, 6]
[1, 4, 2, 5, 3, 6]
True
```

Exercice 6.51 (CodEx : Redimensionner un tableau)

On considère un tableau de R lignes et C colonnes, et l'on souhaite modifier ses dimensions pour obtenir un nouveau tableau de dimensions r lignes et c colonnes. Un tableau est représenté par une liste de listes (les lignes). Les deux tableaux contiennent exactement le même nombre de valeurs, ce qui garantit que $R \times C = r \times c$. Les éléments du nouveau tableau conservent l'ordre de lecture ligne par ligne, de gauche à droite.

Écrire une fonction `redimensionner(T,r,c)` prenant en paramètres le tableau `T` (liste de listes) à redimensionner, les nouvelles dimensions r et c. La fonction renvoie le tableau redimensionné sous forme d'une liste de listes.

Par exemple, le tableau $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$ de taille $R = 4$ et $C = 3$ peut être transformé en le tableau $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{pmatrix}$ de taille $r = 1$ et $c = 12$ ou encore en le tableau $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{pmatrix}$ de taille $r = 2$ et $c = 6$.

Correction

Source : https://codex.forge.apps.education.fr/exercices/redimensionner/

Cet exercice demande de coder une fonction approchant la méthode `reshape` du module `numpy`.

```python
def redimensionner(T, r, c):
    # matrice (de taille R, C) ⇝ liste
    flatten = lambda M: [elem for row in M for elem in row]
    # liste ⇝ matrice de dimension r, c
    reshape = lambda L, r, c: [  L[ i*c : (i+1)*c ] for i in range(r) ]
    # composition
    return reshape(flatten(T), r, c)

# Exemple d'utilisation
original_matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
    [10, 11, 12]
]

R, C = len(original_matrix[0]), len(original_matrix)
print(f"Matrice originale ({R}x{C}) : {original_matrix}")
```

```
r, c = 1, 12
nouvelle_matrice = redimensionner(original_matrix, r, c)
print(f"Matrice redimensionnée ({r}x{c}) : {nouvelle_matrice}")

r, c = 2, 6
nouvelle_matrice = redimensionner(original_matrix, r, c)
print(f"Matrice redimensionnée ({r}x{c}) : {nouvelle_matrice}")
```

```
Matrice originale (3x4) : [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
Matrice redimensionnée (1x12) : [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]]
Matrice redimensionnée (2x6) : [[1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12]]
```

Exercice Bonus 6.52 (Soirée parents-enfants)

Lors d'une soirée, chaque enfant (représenté par une lettre minuscule) doit être accompagné d'au moins un parent (représenté par la même lettre en majuscule). L'objectif est d'afficher les enfants qui sont venus sans parent.

Par exemple, pour la chaîne `'AbaaBBc'`, la fonction devra retourner `'c'`. De même, pour la chaîne `'bbDeaeBfA'`, la fonction devra retourner `'eef'`.

Source : https://twitter.com/riko_schraf/status/1555265499373752321

Correction

La fonction parcourt la chaîne de caractères `s`. Pour chaque caractère en minuscule, elle vérifie si la version en majuscule de ce caractère n'est pas présente dans la chaîne. Si c'est le cas, cela signifie que l'enfant est venu sans parent et le caractère est ajouté à la chaîne de résultats `sol`. Enfin, la fonction renvoie cette chaîne.

```
def soiree(s):
    sol = ""
    for c in s:
        if c.islower() and c.upper() not in s:
            sol = sol+c
    return sol
```

```
# TESTS
for s in [ 'AbaaBBc' , 'bbDeaeBfA' ]:
    print(f"{s = }, {soiree(s) = }")

s = 'AbaaBBc', soiree(s) = 'c'
s = 'bbDeaeBfA', soiree(s) = 'eef'
```

Exercice Bonus 6.53 (Leet speak)

Un hacker remplace les «A» par des «4», les «E» par des «3», les «I» par des «1», les «O» par des «0» et les «S» par des «5». Écrire une fonction qui transforme une chaîne de caractère en majuscules avec les changements indiqués. Par exemple, `hacker("un quizz facile")` donnera `"UN QU1ZZ F4C1L3"`. Pour en savoir plus sur le *leet speak* on pourra consulter la page https://fr.wikipedia.org/wiki/Leet_speak.

Source : https://twitter.com/riko_schraf/status/1555609768898793475

Correction

Input : une chaîne de caractères ; output : une chaîne de caractères.

```
def hacker(s):
    replacements = [('A', '4'), ('E', '3'),
                    ('I', '1'), ('O', '0'),
                    ('S', '5')]
    s = s.upper()
    for old, new in replacements:
        s = s.replace(old, new)
    return s
```

```
# TESTS
for s in [ 'un quizz facile' , "AEIOS"]:
    print( f"hacker('{s}') = {hacker(s)}" )

hacker('un quizz facile') = UN QU1ZZ F4C1L3
hacker('AEIOS') = 43105
```

En version compacte :

```
hacker = lambda s :
↪  s.upper().replace("A","4").replace("E","3").replace("I","1").replace("O","0").replace("S","5")

# TESTS
print(*[ f"hacker('{s}') = {hacker(s)}" for s in ('un quizz facile' , "AEIOS") ],sep="\n")
```

```
hacker('un quizz facile') = UN QU1ZZ F4C1L3
hacker('AEIOS') = 43105
```

Exercice 6.54 (Masquer tickets de caisse)

Écrire une fonction qui masque tous les chiffres d'une chaîne sauf les 4 derniers, comme sur les tickets de caisse. Exemples `masquer("123456")` donnera `"**3456"`; `masquer("789")` donnera `"789"`.

Source : https://twitter.com/riko_schraf/status/1554472875163324416

Correction

Input de type `str`, output de type `str`.

Dans la fonction `masquer`, on commence par vérifier la longueur de la chaîne `s`. Si sa longueur est inférieure ou égale à 4, aucun masquage n'est nécessaire. Dans ce cas, la fonction renvoie simplement la chaîne d'entrée sans aucun changement. Sinon, on renvoie une chaîne construite par concaténation (+) composée de `'*'` répétés un nombre de fois égal à la différence entre la longueur de la chaîne et 4 (pour masquer tous les caractères sauf les 4 derniers) suivie des 4 derniers caractères de la chaîne d'origine.

```python
def masquer(s):
    l = len(s)
    if l<5:
        return s
    else:
        return "*"*(l-4)+s[-4:]
```

```python
# TESTS
for s in ( "123456" , "78" ):
    print(f"{s = }, {masquer(s) = }")
```

```
s = '123456', masquer(s) = '**3456'
s = '78', masquer(s) = '78'
```

6

Exercice Bonus 6.55 (PyDéfis – Les hybrides. Détection de bases inconnues)

Suite au meurtre maquillé du docteur Berube, Mulder retrouve un échantillon dans un tube en verre, sur lequel est indiqué : Purity Control. Intrigué, il remet l'échantillon à Scully qui le fait analyser par le docteur Carpenter. Celle-ci trouve dans l'échantillon une bactérie assez étrange : contrairement à l'ADN de tous les êtres vivants sur Terre, l'ADN de cette bactérie ne contient pas uniquement les quatre bases A, T, G et C, mais aussi d'autres bases a priori inconnues. Le résultat des analyses du docteur Carpenter est transmis dans un fichier contenant sur chaque ligne une partie de la séquence ADN. Chaque morceau d'ADN est donné sous la forme d'une séquence de bases, et dans ce fichier, chaque base est représentée par 4 chiffres binaires :

`0001` pour A, `1010` pour T, `1100` pour G et `0011` pour C.

Voici un morceau d'ADN typique d'un organisme terrestre :

`10100011110000011010`

En effet, en séparant les séquences en groupes de 4 chiffres binaires, on retrouve bien les quatre bases prévues :

```
1010 0011 1100 0001 1010
  T    C    G    A    T
```

Les données d'entrée donnent le contenu du fichier des séquences ADN. Chaque ligne contient le numéro de séquence et la séquence ADN en question. Suite à la disparition soudaine du docteur Carpenter (décidément…), Scully a pour objectif de creuser un peu cette affaire d'ADN. Aidez-la en donnant la liste des numéros des séquences d'ADN qui contiennent entre autres d'autres bases que les quatre bases A, T, G et C.

Si la séquence d'entrée était :

```
N° 1 : 0011001110101100001110100001001111000011110000111100000111001010
N° 2 : 0011000100010001000100110100001110101100110010101100
N° 3 : 0011011000011010000110100100000100010001101011000001101 0
N° 4 : 1100110010100001101000010001000110100011001100010011
N° 5 : 0001101011000001110011000011001100011010101000011100000100110001
```

On pourrait repérer trois bases inconnues dans deux des cinq séquences :

```
N° 2 : 0011000100010001000100110 0100 0011101011001100101011 00
                                 ^^^^

N° 3 : 0011 0110 0001101000011010 0100 0001000100011010110000011010
            ^^^^                  ^^^^
```

Le défi serait alors validé en entrant la séquence : 2, 3 (peu importe l'ordre dans lequel les valeurs sont données).
Source : https://pydefis.callicode.fr/defis/C24_LesHybrides/txt

Exercice 6.56 (CodEx - Nombre de zéros à la fin d'un entier sans `str`)

Écrire une fonction `nb_zeros(n)` qui renvoie le nombre de zéros à la fin de l'écriture décimale de l'entier $n > 0$. On s'interdit, ici, d'utiliser la fonction de conversion `str`. Cette méthode est totalement inefficace avec des nombres très grands. On demande plutôt de compter combien de fois on peut diviser n par 10 avec un reste égal à zéro. Exemples :

a) `nb_zeros(42000)` renvoie 3 b) `nb_zeros(3210)` renvoie 1 c) `nb_zeros(282475249)` renvoie 0
d) `nb_zeros(7**10000)` renvoie 0 e) `nb_zeros(7**10000 * 1000)` renvoie 3

Source : https://codex.forge.apps.education.fr/exercices/nb_zeros/

Correction

```python
def nb_zeros(n):
    # non valable pour n = 0
    cpt = 0
    while n % 10 == 0:
        cpt += 1
        n //= 10
    return cpt
```

```python
# Tests
for n in [42000, 3210, 282475249]:
    print(f"nb_zeros({n}) = {nb_zeros(n)}")

# Pour les tests suivants on ne peut pas afficher
↪ n
# Error : exceeds the limit (4300 digits) for
↪ integer string conversion
# mais on peut calculer le nombre de zéros
print(f"{nb_zeros(7**10000)=}")
print(f"{nb_zeros(7**10000 * 1000)=}")
```

```
nb_zeros(42000) = 3
nb_zeros(3210) = 1
nb_zeros(282475249) = 0
nb_zeros(7**10000)=0
nb_zeros(7**10000 * 1000)=3
```

Exercice 6.57 (Liste qui transforme en 0 les termes à partir du premier 0)

Écrire une fonction `zero` qui prend en entrée une liste de nombres et remplace tous les éléments à partir du premier 0 trouvé par des zéros. Si aucun zéro n'est présent dans la liste, la fonction renvoie la liste inchangée. Exemple : `zero([6,8,0,4,6,3])` donnera `[6,8,0,0,0,0]`.
Source : https://twitter.com/riko_schraf/status/1554156116128440321

Correction

Input de type `list` (et chaque élément de la liste est de type `int` ou `float`), output du même type que l'input.

Voici une solution qui parcourt la liste une seule fois. On initialise une liste `S` de la même longueur que `L` remplie de zéros. Ensuite, tant qu'aucun zéro n'est trouvé, on copie les éléments de `L` dans `S`. Dès qu'un zéro est trouvé, on s'arrête.

```python
def zero(L):
    S = [0]*len(L)
    i = 0
    while i < len(L) and L[i] != 0:
        S[i] = L[i]
        i += 1
    return S
```

Dans le premier test, où un seul zéro est présent dans la liste, les éléments à partir du premier zéro sont remplacés par des zéros. Dans le deuxième, où deux zéros (non contigus) sont présents, on vérifie que c'est bien à partir du premier zéro que les éléments sont mis à zéro. Dans le dernier test, où aucun zéro n'est présent, la liste reste inchangée après l'appel de la fonction `zero`.

```python
for L in ([6, 8, 0, 4, 3, 6], [6, 8, 0, 4, 0, 6], [6, 8, 4, 3, 6]):
    print(f"{L = }, {zero(L) = }")
```

Exercice 6.58 (Password)

Stephan et Sophia ont oublié la sécurité et utilisent des mots de passes simples pour tout. Aide Nikola à développer un module de sécurité de mot de passes. Le mot de passe est considéré comme fort si sa longueur est supérieur ou égale à 10 symboles, il a au moins un chiffre, et doit contenir au moins une lettre majuscule et une lettre minuscule.

- Entrée : une chaîne de caractères contenant uniquement des lettres latines ASCII ou des chiffres.
- Sortie : le booléen `True` si le mot de passe respecte toutes les contraintes, `False` sinon.

Source : https://py.checkio.org/fr/mission/house-password/

Correction

```
password = lambda data : not( len(data)<10 or
                              data.isdigit() or
                              data.isalpha() or
                              data.isupper() or
                              data.islower() )

# TESTS
for p in ['A1213pokl', 'bAse730onE4', 'asasasasasasasaas', 'QWERTYqwerty', '123456123456',
 ↪  'QwErTy911poqqqq']:
⟶print(f"La password {p} est acceptable ? {password(p)}")
```

```
La password A1213pokl est acceptable ? False
La password bAse730onE4 est acceptable ? True
La password asasasasasasasaas est acceptable ? False
La password QWERTYqwerty est acceptable ? False
La password 123456123456 est acceptable ? False
La password QwErTy911poqqqq est acceptable ? True
```

Exercice 6.59 (Distance de Hamming)

La distance de Hamming entre deux chaînes de caractères est le nombre de positions auxquels les caractères correspondants dans les deux chaînes sont différents. Par exemple, la distance de Hamming entre “noiseless” et “nuisances” est 5.

n	o	i	s	e	l	e	s	s
n	u	i	s	a	n	c	e	s
	↑			↑	↑	↑	↑	

Écrire une fonction qui renvoie la distance de Hamming de deux chaînes de caractères (pas forcement de même longueur).

Correction

Input : deux chaînes de caractères de longueur quelconque (en minuscule), output : un entier.

On initialise la distance `d` avec la différence entre la longueur maximale et celle minimale des deux chaînes puis on itère sur la longueur minimale des chaînes pour comparer les caractères. Chaque fois que les caractères diffèrent, on incrémente la distance `d`.

```
def hamming(s1,s2):
⟶l1 = len(s1)
⟶l2 = len(s2)
⟶d = max(l1,l2)-min(l1,l2)
⟶for i in range(min(l1,l2)):
⟶⟶d += s1[i]!=s2[i]
⟶return d

s1 = 'noiseless'
s2 = 'nuisances'
H = hamming(s1, s2)
print(f"La distance de Hamming entre les chaînes '{s1:s}' et '{s2:s}' est {H:d}.")

s1 = 'family'
s2 = 'families'
H = hamming(s1, s2)
print(f"La distance de Hamming entre les chaînes '{s1:s}' et '{s2:s}' est {H:d}.")
```

```
La distance de Hamming entre les chaînes 'noiseless' et 'nuisances' est 5.
La distance de Hamming entre les chaînes 'family' et 'families' est 3.
```

Exercice 6.60 (Nombre de Harshad)

Un nombre Harshad est un nombre entier divisible par la somme de ses chiffres (par exemple, 21 est divisible par $2+1=3$ et est donc un nombre de Harshad).

Écrire d'abord une fonction qui renvoie le booléen `True` ou le booléen `False` si n est un nombre d'Harshad ou non respectivement. Écrire ensuite une fonction lambda qui prend en entrée L, R avec $L < R$ et renvoie la liste des nombres entre L et R (inclus) qui sont de Harshad.

Correction

La première fonction, nommée `is_harshad`, utilise une approche en ligne pour déterminer si un nombre est de Harshad. Elle prend un nombre entier n en entrée, elle convertit ce nombre en une chaîne de caractères à l'aide de `str()`. Ensuite, elle parcourt chaque chiffre de cette chaîne à l'aide d'une compréhension de liste, transformant chaque chiffre en entier `int()`. Elle calcule ensuite la somme de ces chiffres avec `sum()`. Enfin, elle vérifie si le nombre n est divisible par cette somme. Si oui, elle retourne le booléen `False` (autrement dit, si le reste est zéro il faut renvoyer `True` et donc `not 0`).

```
is_harshad = lambda n: not n % sum([int(c) for c in str(n)])
print(is_harshad(201),is_harshad(211))
```

```
True False
```

La deuxième fonction lambda, `list_harshad`, prend deux nombres L et R en entrée et génère une liste des nombres de Harshad compris entre L et R inclus en utilisant une compréhension de liste. Elle utilise la fonction `is_harshad` pour vérifier chaque nombre dans la plage `range(L, R+1)`.

```
list_harshad = lambda L,R: [n for n in range(L,R+1) if is_harshad(n)]
print(list_harshad(10,40))
```

```
[10, 12, 18, 20, 21, 24, 27, 30, 36, 40]
```

Exercice 6.61 (Validité d'un code de photocopieuse)

Le code d'une photocopieuse est un numéro N composé de 4 chiffres. Les codes corrects ont le chiffre le plus à droite égal au reste de la division par 7 de la somme des trois autres chiffres. Ainsi, le code 5739 est incorrect car $5+7+3=15$ et $15 \mod 7 = 1 \neq 9$ tandis que 5731 est correct.

Le but de cet exercice est de créer une fonction qui prend en entrée le code et qui renvoie `"VALIDE"` ou `"NON VALIDE"` (bien noter : on renvoie une chaîne de caractère, on n'utilise pas `print` dans la fonction).

Correction

```
def photocopieuse(N):
    sN = str(N)
    n = sum(int(x) for x in sN[:-1])
    if n%7 == int(sN[-1]):
        return "VALIDE"
    else:
        return "NON VALIDE"
```

```
# TESTS
for N in [5739,5731]:
    print(f"N = {N} {photocopieuse(N)}")
```

```
N = 5739 NON VALIDE
N = 5731 VALIDE
```

En écriture compacte :

```
photocopieuse = lambda N : ["NON VALIDE", "VALIDE"][sum(int(x) for x in str(N)[:-1])%7 ==
↪  int(str(N)[-1])]
```

Notons qu'on peut utiliser `[s0,s1][test]` qui donne `s0` si `test` est `False` (interprété comme 0), `s1` si `test` est `True` (interprété comme 1) :

```
def photocopieuse(N):
    sN = str(N)
    n = sum(int(x) for x in sN[:-1])
    return ["NON VALIDE", "VALIDE"][n%7 == int(sN[-1])]
```

Exercice 6.62 (Numéro ISBN-10)

L'*International Standard Book Number* ISBN-10 est un numéro N à 10 chiffres qui sert à identifier un livre. Le premier chiffre représente le pays, ensuite un bloc de chiffres est attribué à un éditeur, plus un autre bloc est le numéro donné par l'éditeur au livre. Le dernier chiffre est la clé, calculé de telle sorte que si $a_0 a_1 \dots a_9$ désigne un numéro ISBN, alors l'entier

$$\sum_{i=0}^{9} (i+1) a_{9-i}$$

soit divisible par 11.

Créer deux fonctions : une fonction qui vérifie si un code ISBN-10 donné est correct et une fonction qui, pour 9 chiffres donnés, renvoie la clé.

Correction

```
def verific(code):
    s = sum( [(i+1)*int(str(code)[9-i]) for i in range(10)] )
    return s%11==0

def cle(code_sans_cle):
    s=sum( [(i+1)*int(str(code_sans_cle)[9-i]) for i in range(1,10)] )
    return -s%11

# TESTS
for N in [ 2100574223 , 2729812563 ] :
    code_sans_cle = int(str(N)[:-1])
    print(f"verific({N})={verific(N)}. En effet cle({code_sans_cle})={cle(code_sans_cle)}")
```

```
verific(2100574223)=False. En effet cle(210057422)=1
verific(2729812563)=True. En effet cle(272981256)=3
```

Exercice 6.63 (Numéro de sécurité sociale)

Le numéro N de sécurité social est composé de 13 chiffres ($=n$) suivis d'une clé de 2 chiffres ($=c$).

La clé permet de vérifier s'il n'y a pas eu d'erreur en reportant le numéro sur un formulaire ou lors du transfert informatique par exemple, car on doit avoir $c = 97 - r$ où r est le reste de la division euclidienne de n par 97. Par exemple, si le numéro de sécurité social est N = 9 700 000 000 100 94 alors n = 9 700 000 000 100 et le reste de la division euclidienne de n par 97 est 3. La clé est bien $97-3 = 94$. Si on a commis une erreur en écrivant n, il y a peu de chances d'avoir écrit un nombre qui a la même clé.

Le but est de créer une fonction qui prend en entrée le numéro de sécurité sociale et la clé et qui renvoie `"VALIDE"` si la clé correspond au numéro et `"NON VALIDE"` sinon.

Correction

```
def verific(N):
    sN = str(N)
    n = int(sN[:13])
    cle = int(sN[-2:])
    if 97-n%97==cle:
        return "VALIDE"
    else:
        return "NON VALIDE"
```

```
#def verific(N):
#    sN = str(N)
#    n = int(sN[:13])
#    cle = int(sN[-2:])
#    return ["VALIDE", "NON VALIDE"][97-n%97!=cle]
```

```python
# TESTS
for N in [970000000010094,
          123456789101193,
          223456789101193,
          223455789101120]:
    print(f"N={N} {verific(N)}")
```

```
N=970000000010094 VALIDE
N=123456789101193 VALIDE
N=223456789101193 NON VALIDE
N=223455789101120 NON VALIDE
```

Exercice Bonus 6.64 (CheckSum)

Pour réduire la probabilité d'erreurs de transmission d'un code, on introduit un chiffre final (clé) calculé à partir des chiffres précédents. Écrire une fonction `cheksum(N)` qui, pour N donné, renvoie la clé. Les étapes à suivre pour obtenir la clé sont les suivantes :

- notons N le code et c la clé
- soit L la liste des chiffres de N dans l'ordre inverse
- pour les chiffres en position paire, doubler la valeur et en calculer la somme de ses chiffres
- calculer r le reste de la division par 10 de la somme des éléments de L
- si le reste est 0, poser $c = 0$, sinon poser $c = 10 - r$

Exemple : cheksum(992739871) renvoie 3.

Source : https://py.checkio.org/en/mission/check-digit/

Bonus : généraliser au cas d'entrées alphanumériques (combinaison possible de 10 chiffres et 26 lettres majuscules). Pour cela, utiliser la valeur ASCII de chaque caractère. Par exemple : «A» a une valeur ASCII 65 obtenue par `ord("A")`. Pour déterminer sa séquence dans notre cas on devra alors soustraire 48.

Correction

```python
def cheksum(N):
    L = [ int(c) for c in str(N)[::-1] ]
    M = [ ell if i%2==1 else sum( int(x) for x in str(ell*2)) for i,ell in enumerate(L) ]
    r = sum(M)%10
    return 0 if r==0 else 10-r # idem que (10-(r%10))%10

for N in (7992739871,123,61820923155) :
    print( cheksum(N), end=", " )
```

```
3, 0, 3,
```

Bonus

```python
def cheksum(s):
    L = [ ord(c)-48 for c in s[::-1] if c.isnumeric() or c.isalpha() ]
    M = [ ell if i%2==1 else sum( int(x) for x in str(ell*2)) for i,ell in enumerate(L) ]
    r = sum(M)%10
    return 0 if r==0 else 10-r

for s in ("7992739871","123","61820923155","799 273 9871","139-MT","+61 820 9231 55"):
    print( cheksum(s), end=", " )
```

```
3, 0, 3, 3, 8, 3,
```

Exercice 6.65 (Algorithme d'Euclide)

Coder les deux algorithmes suivantes et les valider (voir aussi l'exercice 4.31).

- Algorithme des soustractions successives pour le calcul du **reste de la division euclidienne** de deux entiers :

  ```
  Initialiser a,b in N, b non nul
  Répéter tant que a>=b
      a ← a-b
  fin répéter
  Le reste vaut a
  ```

  Autrement dit, `r=a%b` (mais vous devez coder cet algorithme et ne pas utiliser l'opérateur %).

- Algorithme d'Euclide pour le calcul du **PGCD** de deux entiers :

  ```
  Initialiser a,b in N et a>=b>0
  Répéter tant que r>0
      r ← reste de la division euclidienne de a par b (avec l algorithme précédent)
  ```

```
    ⟶a ← b
    ⟶b ← r
  fin répéter
  Le PGCD vaut a
```

- Variante (Jacob Hacks, 1893)

$$\gcd(a,b) = a+b-ab+2\sum_{i=1}^{b-1}\left\lfloor \frac{ai}{b} \right\rfloor$$

Correction

①
```python
def reste(a,b):
    while a>=b:
        a -= b
    return a

def PGCD(a,b):
    b,a = sorted([a,b]) # ainsi a ≥ b
    while b>0:
        a,b = b,reste(a,b)
    return a

# TESTS
for a,b in [ (2*3,3*5) , (2*3,5*7) , (2*2,2*2*3) ]:
    print(f"PGCD({a},{b})={PGCD(a,b)}")
```

```
PGCD(6,15)=3
PGCD(6,35)=1
PGCD(4,12)=4
```

② Variante

```python
for a,b in [(2*3,3*5),(2*3,5*7),(2*2,2*2*3)]:
    print(f"PGCD({a},{b})={  a+b-a*b+2*sum( int(a*i/b) for i in range(1,b)) }")
```

```
PGCD(6,15)=3
PGCD(6,35)=1
PGCD(4,12)=4
```

③ Pour vérifier, on peut aussi s'appuyer sur la fonction `gcd` du module `math` :

```python
from math import gcd
for a,b in [(2*3,3*5),(2*3,5*7),(2*2,2*2*3)]:
    print(f"PGCD({a},{b})={ gcd(a,b) }")
```

```
PGCD(6,15)=3
PGCD(6,35)=1
PGCD(4,12)=4
```

Exercice Bonus 6.66 (Fusion de listes)

Deux succursales A et B d'une même entreprise ont chacune un fichier relatif aux articles qu'elles ont en stock. Chaque enregistrement est une liste de tuples. Chaque tuple comprend le code d'un article et la quantité disponible en stock pour la succursale concernée. Les deux succursales utilisent les mêmes codes pour les mêmes articles, mais elles n'ont pas obligatoirement les mêmes articles en stock. Par exemple, la liste $A = [(1,10),(2,20)]$ indique que la succursale A a en stock 10 éléments de la référence 1 et 20 éléments de la référence 2 ; la liste $B = [(2,1),(3,2)]$ indique que la succursale B a en stock 1 élément de la référence 2 et 2 éléments de la référence 3.

Les deux succursales décident de fusionner leur stock. Écrire une fonction qui permet de remplacer les deux listes par une seule liste de tuples où chaque tuple contient la référence de l'article et la quantité disponible en stock sur l'ensemble des deux succursales. Dans notre exemple, on devra obtenir la liste $F = [(1,10),(2,21),(3,2)]$.

Correction

Les références sont des entiers qui peuvent être utilisés comme clé d'une liste.

La première méthode prend deux listes de tuples `A` et `B` comme entrée, représentant les stocks de deux succursales. Elle crée une liste `LF` de la taille maximale des références présentes dans les deux succursales. Ensuite, elle itère sur les tuples de `A` et `B` pour ajouter les quantités d'articles ayant la même référence à la même position dans la liste `LF`. Enfin, elle crée une nouvelle liste de tuples en énumérant les éléments de `LF` qui ont une quantité supérieure à zéro.

```
def fusion(A,B):
    max_ref = max([r for r,q in A+B])
    LF = [0 for _ in range(max_ref+1)] # pas optimale car on pourrait allouer une liste très grande même
    ↪ avec une seule référence
    for r,q in A+B :
        LF[r] += q
    return [(r,v) for r,v in enumerate(LF) if v>0]

# TEST
A = [ (1,10), (2,20) ]
B = [ (2,1), (3,2) ]
print(fusion(A,B))
```

```
[(1, 10), (2, 21), (3, 2)]
```

Cependant, cette méthode n'est pas optimale car elle utilise une liste LF basée sur les références des articles, potentiellement de grande taille même si seules quelques références sont effectivement utilisées. Ainsi, cette méthode peut allouer de la mémoire inutilement sur une plage de valeurs étendue, conduisant à une utilisation inefficace des ressources, notamment lorsque les références sont dispersées sur une plage de valeurs très large.

Pour améliorer cette méthode nous allons utiliser des dictionnaires. [1]

```
def fusion(A,B):
    dA, dB = dict(A), dict(B)
    dF = dict(A+B) # si une clé est dans A et B, la valeur dans B écrase celle dans A
    for key in dF:
        if key in dA and key in dB:
            dF[key] = dA[key]+dB[key]
    return [(k,v) for k,v in dF.items()]

# TEST
A = [ (1,10), (2,20) ]
B = [ (2,1), (3,2) ]
print(fusion(A,B))
```

```
[(1, 10), (2, 21), (3, 2)]
```

Exercice 6.67 (Montagnes/vallées : monotonie dans une séquence)

Soit L une liste de nombre donnée de longueur au moins 3. Compléter les fonctions `isMountain(L)/isValley(L)` pour qu'elles renvoient `True` si les éléments de L sont ordonnés de sorte à dessiner une montagne (resp. une vallée), autrement dit s'ils sont ordonnés d'abord dans l'ordre croissant puis décroissant (resp. décroissant puis croissant), `False` sinon.

Correction

La fonction `all()` permet de vérifier une propriété sur toute une séquence d'éléments de manière très concise, sans avoir besoin d'écrire une boucle `for` avec un `if` manuel. Elle prend un itérable (liste, tuple, compréhension de liste, etc.) et renvoie `True` si tous les éléments de l'itérable sont évalués comme `True`, `False` si au moins un élément est `False`.

```
def isMountain(L):
    idx_max = L.index(max(L))
    # Tout avant le maximum doit être croissant
    croissant = all([L[i] < L[i+1] for i in range(idx_max)])
    # Tout après le maximum doit être décroissant
    decroissant = all([L[i] > L[i+1] for i in range(idx_max+1,len(L)-1)])
    return croissant and decroissant

def isValley(L):
    idx_min = L.index(min(L))
    decroissant = all([L[i] > L[i+1] for i in range(idx_min)])
    croissant = all([L[i] < L[i+1] for i in range(idx_min+1,len(L)-1)])
    return decroissant and croissant
```

1. Sujet 20.1 : https://glassus.github.io/terminale_nsi/T6_6_Epreuve_pratique/BNS_2023/?s=03

```
# TESTS
print(isMountain([1, 2, 3, 2, 1])) # renvoie True
print(isMountain([1, 2, 3, 2, 5])) # renvoie False
print(isValley([3, 2, 1, 2, 3]))   # renvoie True
print(isValley([3, 2, 3, 2, 3]))   # renvoie False
```

```
True
False
True
False
```

Exercice 6.68 (Détection de sommets dans un profil de montagne)

Un système d'information géographique représente le profil d'une montagne par une liste L d'entiers relatifs, de longueur au moins 3, telle que deux valeurs consécutives ne soient jamais égales. On appelle *sommet*

- tout élément L_i ($1 \leq i \leq n-2$) tel que $L_{i-1} < L_i > L_{i+1}$;
- l'élément L_0 si $L_0 > L_1$;
- l'élément L_{n-1} si $L_{n-1} > L_{n-2}$.

Compléter la fonction `check_peaks(L)` qui renvoie `True` si le profil contient strictement plus d'un sommet, et `False` sinon.

Source : https://py.checkio.org/fr/mission/mountain-peaks/

Correction

```
def check_peaks(L):
    if len(L) < 3: return False
    peaks = 0
    # sommet à gauche
    if L[0] > L[1]: peaks += 1
    # sommets internes
    for i in range(1, len(L) - 1):
        if L[i-1] < L[i] > L[i+1]: peaks += 1
    # sommet à droite
    if L[-1] > L[-2]: peaks += 1
    return peaks > 1

# TESTS
print(check_peaks([2, 3, 5, 6, 7, 5, 4, 2]))   # False
print(check_peaks([2, 3, 6, 5, 4, 6, 3, 2]))   # True
print(check_peaks([1, 2, 3, 2, 1]))            # False
print(check_peaks([1, 3, 2, 1, 2, 3, 1]))      # True
print(check_peaks([2, 1, 2]))                  # True
```

```
False
True
False
True
True
```

Soit encore

```
def check_peaks(L):
    if len(L) < 3: return False
    # calcul des différences successives
    d = [L[i+1] - L[i] for i in range(len(L)-1)]
    # sommets à gauche et à droite
    peaks = (d[0] < 0) + (d[-1] > 0)
    # sommets internes : changement de signe
    peaks += sum(1 for i in range(len(d)-1) if d[i] > 0 and d[i+1] < 0)
    return peaks > 1

# TESTS
print(check_peaks([2, 3, 5, 6, 7, 5, 4, 2]))   # False
print(check_peaks([2, 3, 6, 5, 4, 6, 3, 2]))   # True
```

```
print(check_peaks([1, 2, 3, 2, 1]))          # False
print(check_peaks([1, 3, 2, 1, 2, 3, 1]))    # True
print(check_peaks([2, 1, 2]))                # True
```

```
False
True
False
True
True
```

Exercice 6.69 (CodEx: Génération de cartes d'anniversaire pour chats)

Lorsqu'un chat souhaite offrir une carte d'anniversaire à un ami chat, il lui offre une carte avec un chat tout mignon et un texte indiquant son âge dans sa vie actuelle. En effet, les chats ont plusieurs vies, ce qui laisse de nombreuses cartes possibles à créer! La société CatCard propose de personnaliser cette carte avec un texte en anglais (`"en"`) ou en français (`"fr"`). Ce texte dépend de l'`age` et de la `num_vie`, deux entiers strictement positifs. Si la langue demandée n'est ni `"fr"` ni `"en"`, la société CatCard propose une réponse standard: `'Je donne ma langue au chat'`.

Écrire une fonction `channiv` qui renvoie le texte demandé. Le texte doit être configuré pour que, si la langue n'est pas précisée, le message soit en français par défaut. Attention à gérer les suffixes ordinaux en français (`1er` pour un âge masculin, `1re` pour une vie féminine) et en anglais (`1st`, `2nd`, `3rd`, `th`).

Exemples:

- `channiv(3, 4)` renvoie `'Joyeux 3e channiversaire de ta 4e vie'`
- `channiv(3, 4, "en")` renvoie `'Happy purrthday, for 3rd year of 4th life'`
- `channiv(1, 1, "fr")` renvoie `'Joyeux 1er channiversaire de ta 1re vie'`
- `channiv(1, 1, "en")` renvoie `'Happy purrthday, for 1st year of 1st life'`
- `channiv(3, 4, "miaou")` renvoie `'Je donne ma langue au chat'`

Source: https://codex.forge.apps.education.fr/exercices/channiv/

Correction

```
def channiv(age, num_vie, lang="fr"):

    def ordinal_fr(n, feminine=False):
        return "1re" if n == 1 and feminine else "1er" if n == 1 else f"{n}e"

    def ordinal_en(n):
        if n == 1: return f"{n}st"
        elif n  == 2: return f"{n}nd"
        elif n == 3: return f"{n}rd"
        else: return f"{n}th"

#    def ordinal_en(n):
#        match n:
#            case 1: return f"{n}st"
#            case 2: return f"{n}nd"
#            case 3: return f"{n}rd"
#            case _: return f"{n}th"

    if lang == "fr":
        return f"Joyeux {ordinal_fr(age)} channiversaire de ta {ordinal_fr(num_vie, feminine=True)} vie"
    elif lang == "en":
        return f"Happy purrthday, for {ordinal_en(age)} year of {ordinal_en(num_vie)} life"
    return "Je donne ma langue au chat"

# Tests
print(channiv(1, 1, "fr"))     # Expected: Joyeux 1er channiversaire de ta 1re vie
print(channiv(3, 4))           # Expected: Joyeux 3e channiversaire de ta 4e vie
print(channiv(3, 4, "en"))     # Expected: Happy purrthday, for 3rd year of 4th life
print(channiv(1, 1, "en"))     # Expected: Happy purrthday, for 1st year of 1st life
print(channiv(3, 4, "miaou")) # Expected: Je donne ma langue au chat
```

```
Joyeux 1er channiversaire de ta 1re vie
Joyeux 3e channiversaire de ta 4e vie
```

```
Happy purrthday, for 3rd year of 4th life
Happy purrthday, for 1st year of 1st life
Je donne ma langue au chat
```

Exercice 6.70 (Rendu monnaie)

La fonction `rendu_monnaie` prend en paramètres deux nombres entiers positifs `somme_due` et `somme_versee` et elle permet de procéder au rendu de monnaie de la différence `somme_versee - somme_due` pour des achats effectués avec le système de pièces de la zone Euro. Toutes les sommes sont exprimées en euros. Les valeurs possibles pour les pièces sont donc $[1,2,5,10,20,50,100,200]$. On utilisera le nombre minimum de pièces.

On utilise pour cela un **algorithme glouton** qui commence par rendre le maximum de pièces de plus grandes valeurs et ainsi de suite. Par la suite, on assimilera les billets à des pièces.

La fonction `rendu_monnaie` renvoie une liste contenant les pièces qui composent le rendu. Ainsi, l'instruction `rendu_monnaie(452, 500)` renvoie la liste des pièeces/billets à rendre $[20,20,5,2,1]$. En effet, la somme à rendre est de $500-452=48$ euros soit $20+20+5+2+1$.

Correction

Exercice 13.2 : https://glassus.github.io/terminale_nsi/T6_6_Epreuve_pratique/BNS_2023/?s=03

```python
def rendu_monnaie(due, versee):
    montant_a_rendre = versee - due
    coupures = [1, 2, 5, 10, 20, 50, 100, 200]
    pieces_rendues = []
    for c in coupures[::-1]:
        q,montant_a_rendre = divmod(montant_a_rendre,c)
        pieces_rendues.extend( [c]*q )
    return pieces_rendues

due, versee = 452, 500
pieces_rendues = rendu_monnaie(due,versee)
print(f"{due = }, {versee = }, {pieces_rendues = }")
```

```
due = 452, versee = 500, pieces_rendues = [20, 20, 5, 2, 1]
```

Exercice 41.2 : https://glassus.github.io/terminale_nsi/T6_6_Epreuve_pratique/BNS_2023/?s=03

Version recursive :

```python
coupures = [200, 100, 50, 20, 5, 2, 1]
def rendu_monnaie(montant_a_rendre, rang):
    if montant_a_rendre == 0:
        return []
    coin = coupures[rang]
    if  coin <= montant_a_rendre :
        return [coin] + rendu_monnaie(montant_a_rendre-coin, rang)
    else:
        return rendu_monnaie(montant_a_rendre, rang+1)

due, versee = 452, 500
pieces_rendues = rendu_monnaie(versee-due,0)
print(f"{due = }, {versee = }, {pieces_rendues = }")
```

```
due = 452, versee = 500, pieces_rendues = [20, 20, 5, 2, 1]
```

Exercice Bonus 6.71 (PyDéfis – Monnaie)

Dans une monnaie imaginaire, vous disposez d'un nombre illimité de pièces de valeur 50, 20, 10, 5, 2 et 1. Vous devez utiliser ces pièces pour rembourser des sommes dues à plusieurs personnes différentes. Pour chaque personne, vous utiliserez le nombre minimum de pièces. L'entrée du problème est constituée de la liste des sommes à rembourser. Vous devez répondre en fournissant une liste de 6 entiers indiquant le nombre de pièces de 50, 20, 10, 5, 2 et 1 à utiliser. Par exemple, si les sommes à payer sont $(43,32,21)$, vous devez répondre $(0,4,1,0,2,2)$.

Si les sommes à payer sont $(77,94,80,67,37,53,61,53,59,3,92,17,44,11,13,75,93,98,91,9)$, quelles pièces de monnaie doit-on utiliser ?

Source : https://pydefis.callicode.fr/defis/Monnaie/t

Exercice 6.72 (Can Balance)

Soit L une liste de nombre. On cherche l'indice de l'élément pivot définit comme suit : le produit scalaire des éléments de la sous-liste à gauche par leur distance à l'élément pivot est égale au produit scalaire des éléments de la sous-liste à droite par leur distance à l'élément pivot. Voici deux exemples :

1. Si $L = [6, 1, 10, 5, 4]$, l'élément pivot est en position $n = 2$. En effet,
   - sous-liste gauche : $L[:n] = [6, 1]$; distances : $[2, 1]$; produit scalaire : $6 \times 2 + 1 \times 1 = 13$;
   - sous-liste droite : $L[n+1:] = [5, 4]$; distances : $[1, 2]$; produit scalaire : $5 \times 1 + 4 \times 2 = 13$.

6	1	10	5	4
↑	↑		↑	↑
2	1		1	2

2. Si $L = [10, 3, 3, 2, 1]$, l'élément pivot est en position $n = 1$. En effet,
   - sous-liste gauche : $L[:n] = [10]$; distances : $[1]$; produit scalaire : $10 \times 1 = 10$;
   - sous-liste droite : $L[n+1:] = [3, 2, 1]$; distances : $[1, 2, 3]$; produit scalaire : $3 \times 1 + 2 \times 2 + 1 \times 3 = 10$.

10	3	3	2	1
↑		↑	↑	↑
1		1	2	3

Correction

https://py.checkio.org/en/mission/can-balance/

Soit p l'indice du pivot et $\ell_i = L[i]$. La distance de l'élément ℓ_i de l'élément pivot ℓ_p est $|i - p|$. Par définition de pivot on a alors

$$\sum_{i=0} (i - p)\ell_i = 0$$

ainsi

$$p = \frac{\sum_{i=0} i\ell_i}{\sum_{i=0} \ell_i}.$$

```python
def can_balance(L):
    p = sum(i*ell for i, ell in enumerate(L)) / sum(L)
    return int(p) if int(p) == p else -1

print(can_balance([6, 1, 10, 5, 4])) # == 2
print(can_balance([6, 1, 10, 5, 4])) # == 2
print(can_balance([10, 3, 3, 2, 1])) # == 1
print(can_balance([7, 3, 4, 2, 9, 7, 4])) # == -1
print(can_balance([42])) # == 0
```

```
2
2
1
-1
0
```

Exercice Bonus 6.73 (Matching Brackets)

Soient `chaine` une chaîne de caractères qui contient des parenthèses (,), des crochets [,], et des guillemets {, }. Compléter la fonction `is_paired(chaine)` qui renvoie `True` si les parenthèse dans `chaine` sont équilibrées, `False` sinon.

Exemple :

- `is_paired( "[]" )` renvoie `True`
- `is_paired( "[)" )` renvoie `False`
- `is_paired( "[()]" )` renvoie `True`
- `is_paired( "[(])" )` renvoie `False`

Correction

```
def is_paired(chaine):
    stack = []
    for c in chaine:
        if c in "({[":
            stack.append(c)
        elif c==")" :
            if len(stack) > 0 and stack[-1]=="(" :
                stack = stack[:-1]
            else:
                return False
        elif c=="]" :
            if len(stack) > 0 and stack[-1]=="[" :
                stack = stack[:-1]
            else:
                return False
        elif c=="}" :
            if len(stack) > 0 and stack[-1]=="{" :
                stack = stack[:-1]
            else:
                return False
    return len(stack) == 0

# TESTS
print( is_paired("{[2*(7-6)+3*(4-5)]/8}") )
print( is_paired("[2+3*(4-5])") )
```

```
True
False
```

Exercice 6.74 (Nombre miroir)

Soit $n \in \mathbb{N}^*$. On appellera "miroir du nombre n" le nombre n écrit de droite à gauche. Par exemple, `miroir(7423)` = 3247. Écrire une fonction qui prend en entrée n et renvoie son miroir (*cf.* exercice 4.72).

Bonus : soit x_1 un nombre à trois chiffres, x_2 son miroir, et supposons que x_1 et x_2 soient des carrés parfaits, autrement dit il existe y_1 et y_2 (à deux chiffres) tels que $y_1^2 = x_1$ et $y_2^2 = x_2$. Afficher tous les nombres x_1 tels que y_2 est le miroir de y_1.

Correction

```
miroir = lambda n : int(str(n)[::-1])
print(miroir(7423))
```

```
3247
```

La partie bonus de l'exercice travaille avec des nombres à trois chiffres. Pour ce faire, on parcourt les chiffres de 1 à 9 pour les centaines (représentés par a), de 0 à 9 pour les dizaines (représentés par b), et de 1 à 9 pour les unités (représentées par c). Ensuite, on forme le nombre à trois chiffres en combinant a, b, et c. On calcule ensuite la racine carrée de ces nombres ($\underline{abc}$ et son miroir $\underline{cba}$) et on vérifie les conditions suivantes :

- la racine carrée est un entier, c'est-à-dire le nombre $\underline{abc}$ (résp. le nombre $\underline{cba}$) est un carré parfait. Pour cela, on vérifie si la racine carrée est un entier avec l'instruction `sqrt_abc==int(sqrt_abc)` (résp. `sqrt_cba==int(sqrt_cba)`) ;
- le miroir de la racine carrée de $\underline{cba}$ est égal à la racine carrée de $\underline{abc}$ (`sqrt_abc==miroir(int(sqrt_cba))`).

Cela garantit que les nombres sont des carrés parfaits et que leurs miroirs sont également des carrés parfaits.

```
for a in range(1,10):
    for b in range(10):
        for c in range(1,10):
            abc = 100*a+10*b+c
            cba = miroir(abc)
            sqrt_abc = abc**0.5
            sqrt_cba = cba**0.5
            if sqrt_abc==int(sqrt_abc) and sqrt_cba==int(sqrt_cba) and sqrt_abc==miroir(int(sqrt_cba)):
                print(f"{abc} car {abc}={int(sqrt_abc)}^2 et {cba}={int(sqrt_cba)}^2")
```

```
121 car 121=11^2 et 121=11^2
144 car 144=12^2 et 441=21^2
169 car 169=13^2 et 961=31^2
441 car 441=21^2 et 144=12^2
484 car 484=22^2 et 484=22^2
961 car 961=31^2 et 169=13^2
```

Remarque : indiquons par $\underline{abc}$ l'écriture décimale du nombre $100a+10b+c$. On remarque qu'un nombre à trois chiffres $\underline{abc}$ satisfait la condition ssi il existe un nombre à deux chiffres $\underline{de}$ tel que $\underline{de}^2 = \underline{abc}$ et $\underline{ed}^2 = \underline{cba}$. Ceci implique les conditions suivantes :

$$\begin{cases}(10d+e)^2 = 100a+10b+c,\\ (10e+d)^2 = 100c+10b+a,\end{cases}$$

$$\implies (10d+e)^2-(10e+d)^2 = 99(a-b) \implies \Big((10d+e)+(10e+d)\Big)\Big((10d+e)-(10e+d)\Big) = 99(a-b)$$

$$\implies \Big(11(d+e)\Big)\Big(9(d+e)\Big) = 99(a-b) \implies 99(d^2-e^2) = 99(a-b) \implies d^2-e^2 = a-b$$

Exercice 6.75 (Point milieu)

On représente un point P du plan de coordonnées (P_x, P_y) par la liste `P=[Px,Py]`. Écrire une fonction qui renvoie le point milieu du segment d'extrémités les points P et Q.

Correction

Le point milieu a pour coordonnées $\left(\dfrac{P_x+Q_x}{2}, \dfrac{P_y+Q_y}{2}\right)$.

```
milieu = lambda P,Q : [ (p+q)/2 for p,q in zip(P,Q) ]

P = [0,0] ; Q = [2,2]
print(milieu(P,Q))
```

```
[1.0, 1.0]
```

Exercice 6.76 (Normaliser une liste)

La *normalisation* d'un ensemble de valeurs est une opération importante en mathématiques et en traitement de données. Elle consiste à transformer *de manière affine* un ensemble de valeurs situées dans un intervalle $[v_{\min}; v_{\max}]$ en un nouvel ensemble de valeurs comprises entre 0 et 1. Autrement dit, on cherche une fonction de la forme $g(x) = mx+q$ telle que

$$g(v_{\min}) = 0 \quad \text{et} \quad g(v_{\max}) = 1.$$

Cette transformation permet de *ramener toutes les valeurs à la même échelle*, ce qui est utile, par exemple, pour comparer des données de natures différentes ou pour préparer des calculs numériques.

1. Écrivez un fonction qui permet de **normaliser** une liste de nombre, c'est-à-dire de transformer toutes ses valeurs pour qu'elles soient comprises entre 0 et 1, *de manière affine*. Par exemple la liste $[30, 60, 75, 130, 40, 100]$ doit être transformée en $[0, 0.3, 0.45, 1, 0.1, 0.7]$.
2. Vérification : commencez par tester la normalisation sur la liste $[1, 0, -1]$. On doit obtenir la liste normalisée $[1, 0.5, 0]$. Ensuite, générez une liste de 10 entiers aléatoires compris entre 10 et 100 avec le code

   ```
   import random
   L = [random.randint(10, 100) for i in range(10)]
   ```

   et assurez-vous que la liste normalisée contient bien au moins un 0 et un 1.

Correction

On cherche une transformation affine

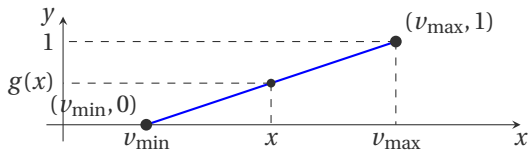
$$g\colon [v_{\min}; v_{\max}] \to [0;1]$$

telle que

$$\begin{cases} g(v_{\min}) = 0,\\ g(v_{\max}) = 1.\end{cases}$$

Il s'agit simplement de l'équation de la droite qui interpole

les deux points $(v_{\min}, 0)$ et $(v_{\max}, 1)$:



$$g(x) = \frac{1}{v_{\max} - v_{\min}}(x - v_{\min}).$$

```
def normaliser(L):
    v_min = min(L)
    v_max = max(L)
    g = lambda x: (x - v_min) / (v_max - v_min)
    return [g(x) for x in L]

# TESTS
import random
for L in ( [1,0,-1] , [random.randint(10,100) for i in range(10)] ):
    print(f"{L = }")
    print(f"{normaliser(L) = }")
```

```
L = [1, 0, -1]
normaliser(L) = [1.0, 0.5, 0.0]
L = [33, 90, 74, 98, 98, 39, 75, 11, 85, 45]
normaliser(L) = [0.25287356321839083, 0.9080459770114943, 0.7241379310344828, 1.0, 1.0,
↪ 0.3218390804597701, 0.735632183908046, 0.0, 0.8505747126436781, 0.39080459770114945]
```

Exercice 6.77 (Couper un intervalle)

Soit $[a; b]$ un intervalle représenté sous la forme d'une liste : `I=[a,b]`.

- Écrire une fonction `Demi(I)` qui scinde l'intervalle $[a; b]$ en deux intervalles $[a; c]$ et $[c; b]$ de même longueur (que vaut c ?) et qui renvoie ces deux intervalles.
  Par exemple, `Demi([0,3])` donnera `[[0, 1.5], [1.5, 3]]`.
- Écrire une fonction `Tiers(I)` qui scinde l'intervalle $[a; b]$ en trois intervalles $[a; c_1]$, $[c_1; c_2]$ et $[c_2; b]$ de même longueur (que valent c_1 et c_2 ?) et qui renvoie ces trois intervalles.
  Par exemple, `Tiers([0,3])` donnera `[[0, 1.0], [1.0, 2.0], [2.0, 3]]`.
- Écrire une fonction `Couper(I,n)` qui scinde l'intervalle $[a; b]$ en n intervalles de même longueur (que vaut cette longueur ?) et qui renvoie ces n intervalles.

Correction

On cherche c tel que $b - c = a - c$ donc $c = \frac{a+b}{2} = a + \frac{b-a}{2}$. Posons $h = \frac{b-a}{2}$, alors $c = a + h$.

```
def Demi(I):
    a = I[0]
    b = I[1]
    c = (a+b)/2
    return [[a,c],[c,b]]

# Test
print(Demi([0,3]))
```

```
[[0, 1.5], [1.5, 3]]
```

On cherche c_1 et c_2 tels que $b - c_2 = a - c_1 = c_2 - c_1$. Posons $h = \frac{b-a}{3}$, alors $c_1 = a + h$ et $c_2 = a + 2h$.

```
def Tiers(I):
    a = I[0]
    b = I[1]
    h = (b-a)/3
    c1 = a+h
    c2 = a+2*h
    return [[a,c1],[c1,c2],[c2,b]]

# Test
print(Tiers([0,3]))
```

```
[[0, 1.0], [1.0, 2.0], [2.0, 3]]
```

Posons $h = \frac{b-a}{n}$ ainsi $a = a + 0h$ et $b = a + nh$. Alors

```python
def Couper(I,n):
    a = I[0]
    b = I[1]
    h = (b-a)/n
    return [[a+i*h,a+(i+1)*h] for i in range(n)]

# Test
print(Couper([0,3],2))
print(Couper([0,3],3))
print(Couper([0,3],6))
```

```
[[0.0, 1.5], [1.5, 3.0]]
[[0.0, 1.0], [1.0, 2.0], [2.0, 3.0]]
[[0.0, 0.5], [0.5, 1.0], [1.0, 1.5], [1.5, 2.0], [2.0, 2.5], [2.5, 3.0]]
```

Exercice 6.78 (Représentation et manipulation de polynômes)

Soit $\mathbb{R}_n[x]$ l'ensemble des polynômes de degré inférieur ou égale à n, $n \in \mathbb{N}$. Tout polynôme de cet espace vectoriel s'écrit de manière unique comme

$$p_n(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + \cdots + a_n x^n, \qquad \text{où } a_i \in \mathbb{R} \text{ pour } i = 0, \ldots n.$$

On peut représenter un polynôme par la liste de ses coefficients et stocker ces $n+1$ valeurs réels $a_0, a_1, \ldots, a_n$ dans une liste (le premier élément de la liste étant le coefficient du terme de degré zéro) : [a]

$$\mathbf{p} = \text{coord}(p_n, \mathscr{C}_n) = [a_0, a_1, a_2, \ldots, a_n] \in \mathbb{R}^{n+1}.$$

Nous utiliserons la liste **p** pour manipuler un polynôme et nous construirons des fonctions pour opérer sur les polynômes à partir de cette représentation. Par exemple, pour construire le polynôme $p_2(x) = 2 - x + x^2$ nous écrirons `p = [2, -1, 1]` et le degré du polynôme est tout simplement `n=len(p)-1`.

1. Implémenter une fonction appelée `eval_pol(p,x)` permettant d'évaluer le polynôme p (la fonction polynomiale) en x où x est une valeur numérique ou une liste. Dans le second cas on doit obtenir une liste contenant les valeurs de la fonction polynomiale aux différents points spécifiés dans la liste **x**. Par exemple, pour évaluer le polynôme $p(x) = 1 + 2x + 3x^2$ en $\mathbf{x} = [-1, 0, 1, 2]$ nous écrirons
```python
p = [1 2 3]
y = eval_pol(p,[-1,0,1,2])
```
et on veut obtenir la liste $\mathbf{y} = p(\mathbf{x}) = [2, 1, 6, 17]$. En effet on a

$$\begin{aligned} p(x) &= 1 + 2x + 3x^2 \\ \mathbf{p} &= \text{coord}[p, \mathscr{C}_2) = [1, 2, 3] \end{aligned}$$

$$\begin{aligned} p(-1) &= 1 + 2 \times (-1) + 3 \times ((-1)^2) = 1 - 2 + 3 = 2 \\ p(0) &= 1 + 2 \times 0 + 3 \times (0^2) = 1 + 0 + 0 = 1 \\ p(1) &= 1 + 2 \times 1 + 3 \times (1^2) = 1 + 2 + 3 = 6 \\ p(2) &= 1 + 2 \times 2 + 3 \times (2^2) = 1 + 4 + 12 = 17 \end{aligned}$$

2. Implémenter une fonction appelée `sum_pol` renvoyant la somme de deux polynômes (attention, si les deux polynômes n'ont pas même degré, il faudra ajouter des zéros en fin du polynôme de plus petit degré afin de pouvoir calculer l'addition des deux vecteurs représentatifs). Par exemple, pour $\mathbf{p} = [1, 2, 3]$ et $\mathbf{q} = [1, -2]$, on veut obtenir $\mathbf{s} = [2, 0, 3]$:

$$\begin{aligned} p(x) &= 1 + 2x + 3x^2 & \mathbf{p} &= \text{coord}(p, \mathscr{C}_2) = [1, 2, 3] \\ q(x) &= 1 - 2x & \mathbf{q} &= \text{coord}(q, \mathscr{C}_1) = [1, -2] \implies \mathbf{q} = \text{coord}(q, \mathscr{C}_2) = [1, -2, 0] \\ s(x) = p(x) + q(x) &= 2 + 3x^2 & \mathbf{s} &= \text{coord}(p + q, \mathscr{C}_2) = [2, 0, 3] \end{aligned}$$

3. Implémenter une fonction appelée `prod_pol` renvoyant le produit de deux polynômes.

Exemple, pour $\mathbf{p} = [1,0,3]$ et $\mathbf{q} = [1,-2]$, on veut obtenir $\mathbf{u} = [1,-2,3,-6]$.

$$\begin{aligned} p(x) &= 1+3x^2 & \mathbf{p} &= \text{coord}(p,\mathscr{C}_2) = [1,0,3] \\ q(x) &= 1-2x & \mathbf{q} &= \text{coord}(q,\mathscr{C}_2) = [1,-2,0] \\ u(x) = p(x)\times q(x) &= 1\times p(x) - 2x\times p(x) = 1-2x+3x^2-6x^3 & \mathbf{u} &= \text{coord}(p\times q,\mathscr{C}_3) = [1,-2,3,-6] \end{aligned}$$

4. Implémenter une fonction appelée `derivee_pol` renvoyant la dérivée d du polynôme p donné en entrée (attention, si $p \in \mathbb{R}_n[x]$, alors $d \in \mathbb{R}_{n-1}[x]$).
   Exemple, pour $\mathbf{p} = [1,2,6]$, on veut obtenir $\mathbf{d} = [2,12]$.

$$\begin{aligned} p(x) &= 1+2x+6x^2 & \mathbf{p} &= \text{coord}(p,\mathscr{C}_2) = [1,2,6] \\ d(x) &= p'(x) = 2+12x & \mathbf{d} &= \text{coord}(d,\mathscr{C}_1) = [2,12] \end{aligned}$$

5. Implémenter une fonction appelée `prim_pol` renvoyant la primitive v du polynôme p donné en entrée ayant 0 pour racine (attention, si $p \in \mathbb{R}_n[x]$, alors $v \in \mathbb{R}_{n+1}[x]$).
   Exemple, pour $\mathbf{p} = [1,2,6]$, on veut obtenir $\mathbf{v} = [0,1,1,2]$.

$$\begin{aligned} p(x) &= 1+2x+6x^2 & \mathbf{p} &= \text{coord}(p,\mathscr{C}_2) = [1,2,6] \\ v(x) &= \int_0^x p(t)\,\mathrm{d}t = \int_0^x 1+2t+6t^2\,\mathrm{d}t = x+x^2+2x^3 & \mathbf{v} &= \text{coord}(v,\mathscr{C}_3) = [0,1,1,2] \end{aligned}$$

6. Implémenter une fonction appelée `int_pol` renvoyant l'intégrale d'un polynôme entre deux valeurs a et b.
   Exemple, pour $\mathbf{p} = [1,2,6]$, $a = 1$ et $b = 2$, on veut obtenir $c = 18$:

$$\begin{aligned} p(x) &= 1+2x+6x^2 \\ c &= \int_a^b p(t)\,\mathrm{d}t = \int_0^b p(t)\,\mathrm{d}t - \int_0^a p(t)\,\mathrm{d}t = v(b)-v(a) = b+b^2+2b^3-a-a^2-2a^3 = 18. \end{aligned}$$

a. Il s'agit des coordonnées de p_n dans la base canonique de l'espace vectoriel $\mathbb{R}_n[x]$, *i.e.* l'ensemble $\mathscr{C}_n = \{1,x,x^2,\dots,x^n\}$. On note cela $\text{coord}(p_n,\mathscr{C}_n) = [a_0,a_1,a_2,\dots,a_n] \in \mathbb{R}^{n+1}$.

Correction

1.
```python
def eval_pol(p,x):
    f = lambda xi,pol : sum( pol[k]*xi**k for k in range(len(pol)) )
    return [ f(xi,p) for xi in x  ]

# TESTS                                          x=[-1, 0, 1, 2], y=[2, 1, 6, 17]
p = [1, 2, 3]
x = [-1,0,1,2]
y = eval_pol(p,x)
print( f"{x=}, {y=}" )
```

◇

2. Sans perte de généralité, supposons que $n > m$, alors

$$\begin{aligned} p(x) &= \sum_{i=0}^{n} a_i x^i = \sum_{i=0}^{m} a_i x^i + \sum_{i=m+1}^{n} a_i x^i & \text{coord}(p,\mathscr{C}) &= [a_0,a_1,a_2,\dots,a_m,a_{m+1},\dots,a_n] \\ q(x) &= \sum_{i=0}^{m} b_i x^i = \sum_{i=0}^{m} b_i x^i + \sum_{i=m+1}^{n} 0\times x^i & \text{coord}(q,\mathscr{C}) &= [b_0,b_1,b_2,\dots,b_m] \\ (p+q)(x) &= \sum_{i=0}^{m} (a_i+b_i)x^i + \sum_{i=m+1}^{n} a_i x^i & \text{coord}(p+q,\mathscr{C}) &= [a_0+b_0,a_1+b_1,a_2+b_2,\dots,a_m+b_m,a_{m+1},\dots,a_n] \end{aligned}$$

```python
def sum_pol(p,q):
    S = [0 for _ in range(max(len(p),len(q)))]
    for i in range(len(p)):
        S[i] += p[i]
    for i in range(len(q)):
        S[i] += q[i]
    return S
```

```
# TESTS
p, q = [1, 2, 3] , [1,-2]
print( f"{p=}, {q=}, s={sum_pol(p,q)}" )
```

```
p=[1, 2, 3], q=[1, -2], s=[2, 0, 3]
```

◇

3. Produit de deux polynômes :

```
def prod_pol(p,q):
  result = [0 for _ in range(len(p) + len(q))]
  for i in range(len(p)):
    for j in range(len(q)):
      result[i+j] += p[i] * q[j]
  return result
```

```
# TESTS
p, q = [1, 0, 3] , [1, -2]
print( f"{p=}, {q=}, s={prod_pol(p,q)}" )
```

```
p=[1, 0, 3], q=[1, -2], s=[1, -2, 3, -6, 0]
```

◇

4. Remarquons que

$$p(x) = \sum_{i=0}^{n} a_i x^i \qquad d(x) = p'(x) = \sum_{i=0}^{n} i\, a_i x^{i-1}$$

$$\mathrm{coord}(p, \mathscr{C}_n) = [a_0, a_1, a_2, \ldots, a_n] \qquad \mathrm{coord}\,(d, \mathscr{C}_{n-1}) = [a_1, 2a_2, \ldots, n a_n]$$

```
def derivee_pol(p):
    n = len(p)
    return [ i*p[i] for i in range(1,n)  ]
```

```
# TESTS
p=[1];        print(f"{p=}, d={derivee_pol(p)}")
p=[1,2];      print(f"{p=}, d={derivee_pol(p)}")
p=[1,2,6];    print(f"{p=}, d={derivee_pol(p)}")
p=[4,5,0,2]; print(f"{p=}, d={derivee_pol(p)}")
```

```
p=[1], d=[]
p=[1, 2], d=[2]
p=[1, 2, 6], d=[2, 12]
p=[4, 5, 0, 2], d=[5, 0, 6]
```

◇

5. Remarquons que

$$p(x) = \sum_{i=0}^{n} a_i x^i \qquad v(x) = \int_0^x p(t)\,\mathrm{d}t = \sum_{i=0}^{n} a_i \int_0^x t^i\,\mathrm{d}t = \sum_{i=0}^{n} a_i \frac{x^{i+1}}{i+1}$$

$$\mathrm{coord}(p, \mathscr{C}_n) = [a_0, a_1, a_2, \ldots, a_n] \qquad \mathrm{coord}\,(v, \mathscr{C}_{n+1}) = \left[0, \frac{a_0}{0+1}, \frac{a_1}{1+1}, \frac{a_2}{2+1}, \ldots, \frac{a_n}{n+1}\right]$$

```
def prim_pol(p):
    n = len(p)
    return [0] + [ p[i]/(i+1) for i in range(n)  ]
```

```
# TESTS
p=[1];        print(f"{p=}, d={prim_pol(p)}")
p=[1,2];      print(f"{p=}, d={prim_pol(p)}")
p=[1,2,6];    print(f"{p=}, d={prim_pol(p)}")
p=[1,2,1,1]; print(f"{p=}, d={prim_pol(p)}")
```

```
p=[1], d=[0, 1.0]
p=[1, 2], d=[0, 1.0, 1.0]
p=[1, 2, 6], d=[0, 1.0, 1.0, 2.0]
p=[1, 2, 1, 1], d=[0, 1.0, 1.0,
↪  0.3333333333333333, 0.25]
```

◇

6.
```
def int_pol(p,a,b):
    prim = prim_pol(p)
    L = eval_pol(prim,[a,b])
    return L[1]-L[0]
```

```
# TESTS
p, a, b = [1,2,6] , 1, 2
print(f"{int_pol(p,a,b)=}")
```

```
int_pol(p,a,b)=18.0
```

Exercice 6.79 (Triangle de Pascal)

On cherche à déterminer les valeurs du triangle de Pascal :

$$\begin{array}{cccccc} 1 & & & & & \\ 1 & 1 & & & & \\ 1 & 2 & 1 & & & \\ 1 & 3 & 3 & 1 & 1 & \\ 1 & 4 & 6 & 4 & 1 & \\ 1 & 5 & 10 & 10 & 5 & 1 \end{array}$$

Dans ce triangle, chaque ligne commence et se termine par le nombre 1. De plus, on additionne deux valeurs successives d'une ligne pour obtenir la valeur qui se situe sous la deuxième valeur.

Compléter la fonction `pascal` prenant en paramètre un entier $n \geq 2$. Cette fonction doit renvoyer une liste de listes correspondante au triangle de Pascal de la ligne 0 à la ligne n (incluse).

Correction

① Version du sujet 17.2 : https://glassus.github.io/terminale_nsi/T6_6_Epreuve_pratique/BNS_2023/?s=03

```
def pascal(n):
    triangle = [[1]]
    for k in range(1, n+1):
        ligne_k = [1]
        for i in range(1, k):
            ligne_k.append(triangle[k-1][i-1] + triangle[k-1][i])
        ligne_k.append(1)
        triangle.append(ligne_k)
    return triangle

T = pascal(4)
# print(T)
# affichage amelioré
from tabulate import tabulate
print(tabulate(T,tablefmt='plain'))
```

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
```

② Sachant que $p_{ki} = \binom{k}{i}$ on peut écrire

```
from math import comb
pascal = lambda n : [ [comb(k,i) for i in range(k+1)] for k in range(n+1)  ]
T = pascal(4)
from tabulate import tabulate
print(tabulate(T,tablefmt='plain'))
```

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
```

Exercice 6.80 (Voyelles)

Écrire une fonction qui prend en entrée une chaîne de caractères donnée et renvoie le nombre de voyelles.

Correction

```
VOYELLES_FR = "AaÀàEeÈèÉéÊêËëIiÎîÏïOoÖöUuÙùYy"

#def voyelles(s):
#⟶return len([x for x in s if x in VOYELLES_FR])

voyelles = lambda s : len([x for x in s if x in VOYELLES_FR])

# TESTS
for s in ["citron", "fraise", "aïoli", "pèche", "Supercalifragilisticexpidélilicieux" ]:
⟶print(f"{s} contient {voyelles(s)} voyelles")
```

```
citron contient 2 voyelles
fraise contient 3 voyelles
aïoli contient 4 voyelles
pèche contient 2 voyelles
Supercalifragilisticexpidélilicieux contient 16 voyelles
```

Exercice 6.81 (CodEx: Élocution chez le dentiste)

Chez le dentiste, la bouche grande ouverte, lorsqu'on essaie de parler, il ne reste que les voyelles. Même les ponctuations sont supprimées. Écrivez une fonction `dentiste` qui prend en paramètre une chaîne de caractères `texte` et qui renvoie une autre chaîne ne contenant que les voyelles de `texte`, placées dans le même ordre que dans `texte`.

Source: https://codex.forge.apps.education.fr/exercices/dentiste/

Correction

```
VOYELLES = "AaÀàEeÈèÉéÊêËëIiÎîÏïOoÖöUuÙùYy"

def dentiste(texte):
    resultat = ""
    for lettre in texte:
        if lettre in VOYELLES:
            resultat += lettre
    return resultat

# dentiste = lambda texte : ''.join([c for c in texte if c in voyelles])

# TESTS
print(dentiste("Bonjour, comment ça va ?"))
print(dentiste("La voiture est rouge."))
print(dentiste("Ceci est un test !"))
```

```
oouoeaa
aoiueeoue
eieue
```

Exercice 6.82 (Pangramme)

Un pangramme est une phrase comportant au moins une fois chaque lettre de l'alphabet. Écrire une fonction qui prend en entrée une chaîne de caractères `s` donnée et renvoie `True` si elle contient toutes les lettres de l'alphabet (*i.e.* `s` est un pangramme [a]), `False` sinon.

On prendra comme alphabet `"abcdefghijklmnopqrstuvwxyz"`.

a. Un pangramme contient toutes les lettres de l'alphabet. Deux exemples classiques:
- "Portez ce vieux whisky au juge blond qui fume"
- "The quick brown fox jumps over the lazy dog"

Plus fort encore, ce pangramme de Gilles Esposito-Farèse, qui contient toutes les lettres accentuées et ligatures du français: "Dès Noël où un zéphyr haï me vêt de glaçons würmiens, je dîne d'exquis rôtis de bœuf au kir à l'aÿ d'âge mûr et cætera!"

Correction

```
def pangramme(s):
⟶s = s.lower()
```

```
    alphabet="abcdefghijklmnopqrstuvwxyz"
    for x in alphabet:
        if x not in s:
            print(f"Il manque la lettre {x}")
            return False
    return True

# TESTS
print(pangramme("portez ce vieux whisky au juge blond qui fume"))
print(pangramme("portez ce vieux whisky au juge blond qui boit"))
print(pangramme("Monsieur Jack vous dactylographiez bien mieux que votre ami Wolf"))
print(pangramme("Buvez de ce whisky que le patron juge fameux"))
```

```
True
Il manque la lettre f
False
True
True
```

Exercice 6.83 (Stock Profit)

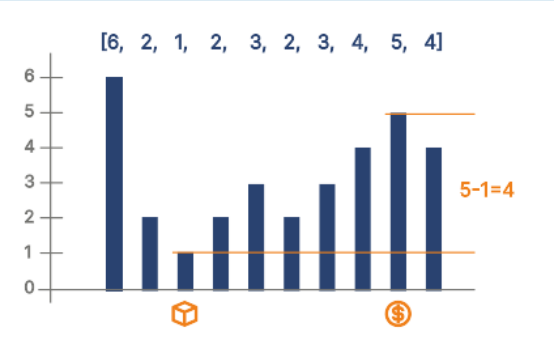
En tant que courtier en bourse se concentrant sur une seule transaction, vous cherchez à maximiser votre profit en achetant une action à un prix inférieur et en la revendant plus tard à un prix supérieur au cours d'une période donnée. Votre fonction doit analyser les fluctuations de prix de l'action au fil du temps. Elle doit identifier l'opportunité la plus rentable en calculant le bénéfice potentiel maximal réalisable dans le cadre de ces fluctuations. Cela signifie qu'il faut trouver le prix le plus élevé pour vendre l'action **après** l'avoir achetée au prix le plus bas possible. Toutefois, s'il n'y a aucune possibilité de réaliser un bénéfice, par exemple lorsque le prix de l'action diminue constamment ou reste inchangé, la fonction doit renvoyer zéro, ce qui indique qu'il n'y a pas d'opportunité viable pour une transaction rentable.

Données d'entrée : prix des actions sous forme de liste d'entiers.

Résultat de sortie : bénéfice maximal possible sous forme d'un nombre entier.

Exemples :

- `stock_profit([6, 2, 1, 2, 3, 2, 3, 4, 5, 4])` renvoie 4 (voir la figure ci-contre)
- `stock_profit([2, 3, 4, 5])` renvoie 3
- `stock_profit([3, 1, 3, 4, 5, 1])` renvoie 4
- `stock_profit([4, 3, 2, 1])` renvoie 0



Source : https://py.checkio.org/fr/mission/stock-profit/

Correction

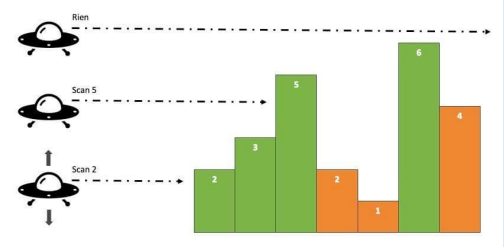
```
stock_profit = lambda stock : max([ max(stock[i:])- s for i,s in enumerate(stock) ])

# TESTS
T = []
T.append( [6, 2, 1, 2, 3, 2, 3, 4, 5, 4] )
T.append( [2, 3, 4, 5] )
T.append( [3, 1, 3, 4, 5, 1] )
T.append( [4, 3, 2, 1] )
T.append( [60, 50, 51, 52, 40] )
for L in T:
    print(f"stock_profit({L})={stock_profit(L)}")
```

```
stock_profit([6, 2, 1, 2, 3, 2, 3, 4, 5, 4])=4
stock_profit([2, 3, 4, 5])=3
stock_profit([3, 1, 3, 4, 5, 1])=4
stock_profit([4, 3, 2, 1])=0
stock_profit([60, 50, 51, 52, 40])=2
```

Exercice 6.84 (OVNI)

Un OVNI se déplace à la verticale et scanne horizontalement les immeubles devant lui.L'objectif est de déterminer quels immeubles seront touchés par le scanner.

Par exemple, si les hauteurs des immeubles sont données par la liste `[2, 3, 5, 2, 1, 6, 4]`, alors les immeubles touchés par le scanner seront ceux de hauteurs `[2, 3, 5, 6]` (voir la figure ci-contre). Dans cet exemple, l'OVNI commence par scanner le premier immeuble de hauteur 2. Ensuite, il scanne l'immeuble de hauteur 3, puis l'immeuble de hauteur 5, et enfin celui de hauteur 6. Les immeubles de hauteurs 2 et 1 ne seront pas touchés par le scanner car ils sont plus petits que les immeubles voisins.



Source: https://twitter.com/riko_schraf/status/1554717715012673536

Correction

```python
def scan(L):
    S = [L[0]]
    for ell in L[1:]:
        if ell>S[-1]:
            S.append(ell)
    return S
```

```python
# TESTS
for L in ( [2,3,5,2,1,6,4] , [3,1,0,2] ):
    print(f"{L = }, {scan(L) = }")
```

```
L = [2, 3, 5, 2, 1, 6, 4], scan(L) = [2, 3, 5, 6]
L = [3, 1, 0, 2], scan(L) = [3]
```

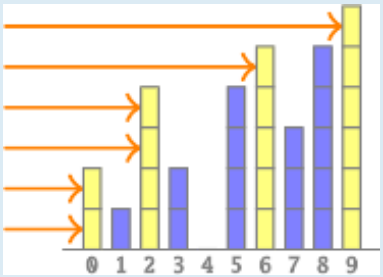
Exercice 6.85 (CodEx: Bâtiments éclairés au soleil couchant)

Lorsque des bâtiments sont alignés, ils se font de l'ombre les uns les autres. Dans cet exercice, nous sommes au soleil couchant, les rayons du soleil sont donc supposés horizontaux et viennent de la gauche du paysage.

Le schéma ci-contre illustre un soleil couchant qui éclaire 9 bâtiments, les rayons du soleil sont représentés par des flèches horizontales.

- Les bâtiments aux indices 0 et 2 reçoivent des rayons de soleil alors que le bâtiment à l'indice 1 est masqué.
- Les 4 bâtiments aux indices $[0,2,6,9]$ reçoivent des rayons de soleil sur au moins un étage et sont donc éclairés, alors que les autres ne le sont pas.
- Il n'y a pas de bâtiment à l'indice 4.

La hauteur des bâtiments (en nombre d'étages) est donnée par une liste d'entiers positifs. Une hauteur de zéro étage signifie l'absence de bâtiment. Pour l'exemple ci-contre, cette liste est `[2, 1, 4, 2, 0, 4, 5, 3, 5, 6]`.



Écrire une fonction `nb_batiments_eclaires` qui prend en argument la liste `hauteurs` des bâtiments et qui renvoie le nombre de bâtiments éclairés.

Source: https://codex.forge.apps.education.fr/exercices/soleil_couchant/

Correction

```python
def nb_batiments_eclaires(L):
    """Soleil couchant : les rayons arrivent de la gauche."""
    tot = 0
    h_maxi = 0
    for h_bat in L:
```

```
        if h_bat>h_maxi:
            h_maxi = h_bat
            tot += 1
    return tot

# TESTS
for L in ( [2, 1, 4, 2, 0, 4, 5, 3, 5, 6] , [0, 3, 1, 2] ):
    print(f"{L = }, {nb_batiments_eclaires(L) = }")
```

```
L = [2, 1, 4, 2, 0, 4, 5, 3, 5, 6], nb_batiments_eclaires(L) = 4
L = [0, 3, 1, 2], nb_batiments_eclaires(L) = 1
```

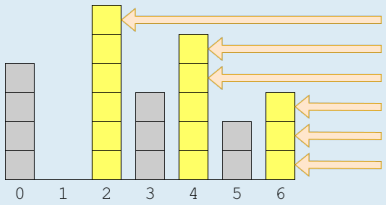
Exercice 6.86 (CodEx: Bâtiments éclairés au soleil levant)

Lorsque des bâtiments sont alignés, ils se font de l'ombre les uns les autres. Dans cette variante, nous sommes au soleil levant: les rayons du soleil arrivent horizontalement par la droite du paysage.

Le schéma ci-contre illustre cette situation: seuls les bâtiments qui ne sont pas masqués par un bâtiment plus haut placé à leur droite reçoivent effectivement la lumière.

- Les bâtiments aux indices 6, 4 et 2 sont éclairés: ils reçoivent au moins un rayon.
- Le bâtiment d'indice 5 est masqué par celui d'indice 6.
- Une hauteur égale à 0 signifie l'absence de bâtiment.

Pour l'exemple représenté, la liste des hauteurs est [4, 0, 6, 3, 5, 2, 3]..



Écrire une fonction `nb_batiments_eclaires` qui prend en argument la liste `hauteurs` des bâtiments et renvoie le nombre de bâtiments éclairés par un soleil levant.

Source: https://codex.forge.apps.education.fr/exercices/soleil_levant/

Correction

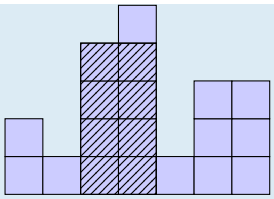
```
def nb_batiments_eclaires(hauteurs):
    """Soleil levant : les rayons arrivent de la droite."""
    tot = 0
    h_maxi = 0
    # On parcourt de droite à gauche
    for h in hauteurs[::-1]:
        if h > h_maxi:
            h_maxi = h
            tot += 1
    return tot

# TESTS
print(nb_batiments_eclaires([4, 0, 6, 3, 5, 2, 3]))  # → 3
print(nb_batiments_eclaires([4, 3, 2, 1, 0]))        # → 4
```

```
3
4
```

Exercice 6.87 (Plus grande surface rectangulaire dans un histogramme)

Étant donné un histogramme représenté par une liste non vide d'entiers, par exemple `[2, 1, 4, 5, 1, 3, 3]`, écrire une fonction qui renvoie l'aire maximale d'un rectangle pouvant être formé à l'aide des barres de l'histogramme.

Dans l'exemple, la surface maximale est de 8 unités.



Correction
Source : https://py.checkio.org/en/mission/largest-histogram/

```
def largest_histogram(histo):
  n = len(histo)
  mymax = 1
  if n==1:
    return histo[0]
  for i in range(n):
    for j in range(i,n):
      mymax = max(mymax,(j+1-i)*min(histo[i:j+1]))
  return mymax
```

```
# TESTS
TESTS = [
          [2, 1, 4, 5, 1, 3, 3],   # sol = 8
          [31, 96],                # sol = 96
          [31, 61]                 # sol = 31*2
        ]
for H in TESTS:
    print(largest_histogram(H))
```

```
8
96
62
```

Cette approche fonctionne pour des histogrammes de petite taille, mais on peut l'améliorer. En effet, il est en $O(n^2)$, ce qui devient lent pour des histogrammes de grande taille. L'approche avec une pile permet de résoudre le problème en $O(n)$.

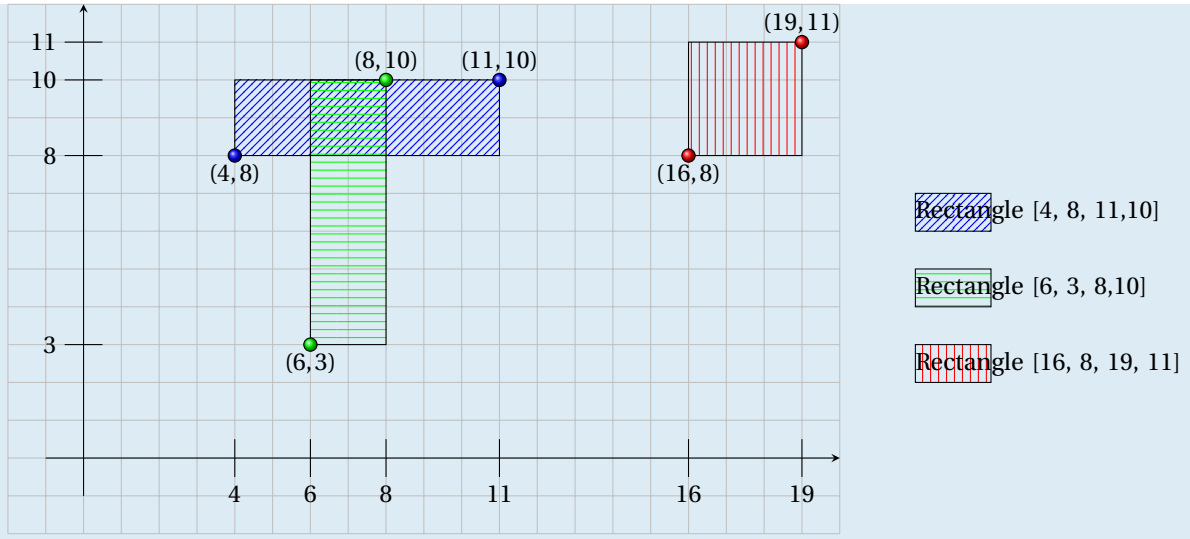
```
def largest_histogram(histo):
    histo.append(0)  # barre sentinelle pour vider la pile à la fin
    stack = []
    max_area = 0
    for i, h in enumerate(histo):
        while stack and histo[stack[-1]] > h:
            height = histo[stack.pop()]
            width = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, height * width)
        stack.append(i)
    histo.pop()  # restaurer la liste initiale
    return max_area


for H in TESTS:
    print(largest_histogram(H))
```

```
8
96
62
```

Exercice 6.88 (Calcul de l'aire d'union de rectangles superposés)

Calculer la surface couverte par l'union de plusieurs rectangles. Chaque rectangle est représenté par 4 entiers : les deux premiers entiers indiquent les coordonnées du coin inférieur gauche et les deux suivants indiquent les coordonnées du coin supérieur droit. L'entrée est fournie sous la forme d'une liste de listes où chaque liste représente un rectangle. Les rectangles peuvent se chevaucher, ce qui signifie que la simple addition des surfaces de chaque rectangle ne donnera pas la surface totale correcte. Au lieu de cela, les régions qui se chevauchent ne doivent être comptées qu'une seule fois.

Exemple :



Étant donné les rectangles [4 8 11 10 ; 6 3 8 10 ; 16 8 19 11], la surface couverte par la réunion de ces rectangles est 33.

Correction

Source : https://py.checkio.org/en/mission/rectangles-union/

```
# D'après l'énoncé, les coordonnées sont des entiers
def rectangles_union(recs):
    area_of_union = len(set((x,y) for x1, x2, y1, y2 in recs for x in range(x1,y1) for y in
    ↪  range(x2,y2)))
    return area_of_union

# # Cas général
# def rectangles_union(recs):
#     xs = sorted(sum(recs, [])[::2])
#     ys = sorted(sum(recs, [])[1::2])
#     boxes = [(xs[i], ys[j], xs[i+1], ys[j+1])
#              for i in range(len(xs)-1)
#              for j in range(len(xs)-1)]
#     area_of_union = 0
#     for (x1, y1, x2, y2) in boxes:
#         for (x3, y3, x4, y4) in recs:
#             if x3 <= x1 <= x4 and x3 <= x2 <= x4 and \
#                y3 <= y1 <= y4 and y3 <= y2 <= y4:
#                    area_of_union += abs(x1-x2)*abs(y1-y2)
#                    break
#     return area_of_union


# TESTS
R1 = [4, 8, 11, 10]
R2 = [6, 3, 8, 10]
R3 = [16, 8, 19, 11]

print(f"R1 : {rectangles_union([R1])}")
print(f"R2 : {rectangles_union([R2])}")
print(f"R3 : {rectangles_union([R3])}")
print(f"R1 et R3 (disjoint) : {rectangles_union([R1, R3])}")
print(f"R1 et R2 (overlap) : {rectangles_union([R1, R2])}")
print(f"R1, R2 et R3 : {rectangles_union([R1, R2, R3])}")
```

```
R1 : 14
R2 : 14
R3 : 9
R1 et R3 (disjoint) : 23
R1 et R2 (overlap) : 24
R1, R2 et R3 : 33
```

Exercice 6.89 (Nombres premiers)

Un nombre premier est un entier naturel qui admet exactement deux diviseurs distincts entiers et positifs : 1 et lui-même. Ainsi 1 n'est pas premier, mais 2 oui.

Écrire la liste des nombres premiers compris entre 1 et 100.

Remarques :

- Pour déterminer si un nombre n est premier, on va chercher s'il existe un nombre différent de 1 et de n qui le divise ;
- un diviseur de n (différent de 1 et de n) est forcement inférieur à n ;
- si n n'est pas premier, il est divisible par un nombre inférieur ou égal à la racine entière de n.

Pour savoir si le nombre n est premier, il suffit alors de calculer les restes des divisions de n par 2, puis 3, puis 4... jusqu'à l'arrondi par défaut de $\sqrt{n}$. Si un des restes est nul, le nombre n'est pas premier. Si aucun reste n'est nul, le nombre n est premier.

Correction

```
def isPrime(n):
    if n <= 3: return n>1
    if n%2 == 0 or n%3 == 0 : return False
    for i in range(3,int(n**0.5)+1,2):    # on s'arrête à √n
        if n%i==0: return False
    return True
print([i for i in range(2,101) if isPrime(i)])
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Exercice Bonus 6.90 (Jumeaux)

Les nombres premiers jumeaux sont des nombres premiers qui diffèrent de 2, par exemple 11 et 13 ou 41 et 43. Écrivez une fonction qui renvoie une liste triée de tuples de nombres premiers jumeaux compris entre `lowVal` et `highVal`. Pour déterminer si un nombre n est premier, voir l'exercice 6.89.

Correction

```
def isPrime(n):
    if n <= 3: return n>1
    if n%2 == 0 or n%3 == 0 : return False
    for i in range(3,int(n**0.5)+1,2):    # on s'arrête à √n
        if n%i==0: return False
    return True

def twin_primes(lowVal, highVal):
    all_prime = [n for n in range(lowVal, highVal+1) if isPrime(n)]
    twin_prime_list = []
    for i in range(len(all_prime)-1):
        if all_prime[i+1]-all_prime[i] == 2:
            twin_prime_list.append((all_prime[i],all_prime[i+1]))
    return twin_prime_list

# Example usage:
lowVal = 1
highVal = 43
result = twin_primes(lowVal, highVal)
print(result)
```

```
[(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43)]
```

Exercice Bonus 6.91 (Défi Turing n°96 – Projet d'Euler n°87)

28 est le plus petit nombre que l'on peut écrire comme $p^2 + q^3 + r^4$, avec p, q et r premiers.

$$28 = 2^2 + 2^3 + 2^4$$
$$33 = 3^2 + 2^3 + 2^4$$
$$49 = 5^2 + 2^3 + 2^4$$
$$47 = 2^2 + 3^3 + 2^4$$

Combien de nombres inférieurs ou égal à 1 milliard peuvent s'écrire de cette façon ?

Correction

```
def isPrime(n):
    if n <= 3: return n>1
    if n%2 == 0 or n%3 == 0 : return False
    for i in range(3,int(n**0.5)+1,2):    # on s'arrête à √n
        if n%i==0: return False
    return True

def count_numbers(limit):
    primes = [n for n in range(2, int(limit ** 0.5) + 1) if isPrime(n)]
    count = set()
    for p in primes:
        p2 = p ** 2
        for q in primes:
            q3 = q ** 3
            if p2 + q3 >= limit:
                break
            for r in primes:
                r4 = r ** 4
                num = p2 + q3 + r4
                if num <= limit:
                    count.add(num)
                else:
                    break
    return len(count)

# limit = 50*10**6 # Euler
limit = 10**9 # Turing
result = count_numbers(limit)
print("Le nombre de nombres inférieurs ou égaux à 1 milliard pouvant s'écrire comme p^2 + q^3 + r^4 avec
↪ p, q, et r premiers est :", result)
```

```
Le nombre de nombres inférieurs ou égaux à 1 milliard pouvant s'écrire comme p^2 + q^3 + r^4 avec p, q,
↪ et r premiers est : 16888551
```

Exercice Bonus 6.92 (PyDéfis – Premier particulier (1))

Un nombre est premier s'il a exactement 2 diviseurs, 1 et lui même. D'autre part, il existe une infinité de nombres premiers de la forme $34k + 35$ (avec $k \geq 0$).

L'entrée du problème est un nombre entier n. Vous devez répondre en donnant le n-ème nombre premier de la forme $34k + 35$ ainsi que la valeur de k pour laquelle on l'obtient.

Testez votre code : par exemple, si nous prenions les nombres premiers de la forme $v(k) = 5k + 7$ et cherchions le troisième (entrée du problème $n = 3$), les réponses seraient 37 et 6. En effet $v(0) = 7$ est premier, $v(1) = 12$ n'est pas premier, $v(2) = 17$ est premier, $v(3) = 22$, $v(4) = 27$, $v(5) = 32$ ne sont pas premiers, et $v(6) = 37$ est le troisième nombre premier obtenu. Pour cet exemple, il faudrait donc répondre $37, 6$.

Source : https://pydefis.callicode.fr/defis/PremierParticulier1/txt

Exercice Bonus 6.93 (Pydéfi – Einstein)

Un nombre d'Einstein E est tel que $E = mc^2$ où m et c sont des nombres premiers différents.

Par exemple, 68 est un nombre d'Einstein car il peut s'écrire sous la forme $68 = 17 \times 2^2$ où 2 et 17 sont des nombres premiers.

Défi : trouvez les 4 premiers nombres d'Einstein impairs consécutifs (la différence entre deux nombres impairs consécutifs est 2).

Source : https://pydefis.callicode.fr/defis/Einstein/txt

Exercice Bonus 6.94 (Mot parfait)

On affecte à chaque lettre de l'alphabet un code selon la position dans l'alphabet : A $\rightsquigarrow$ 1, B $\rightsquigarrow$ 2, ..., Z $\rightsquigarrow$ 26. Pour un mot donné, on détermine d'une part son *code alphabétique concaténé*, obtenu par la juxtaposition des codes de chacun de ses caractères, et d'autre part, *son code additionné*, qui est la somme des codes de chacun de ses caractères.

Par ailleurs, on dit que ce mot est «*parfait*» si le code additionné divise le code concaténé.

Exemples :

- Pour le mot `"PAUL"`,
  - le code concaténé est la chaîne `"1612112"`, soit l'entier 1612112 ;
  - son code additionné est l'entier 50 car $16 + 1 + 21 + 12 = 50$;
  - le mot `"PAUL"` n'est pas parfait car 50 ne divise pas l'entier 1612112.
- Pour le mot `"ALAIN"`,
  - le code concaténé est la chaîne `"1121914"`, soit l'entier 1121914 ;
  - son code additionné est l'entier 37 car $1 + 12 + 1 + 9 + 14 = 37$;
  - le mot `"ALAIN"` est parfait car 37 divise l'entier 1121914.

Compléter la fonction `est_parfait` qui prend comme argument une chaîne de caractères `mot` (en lettres majuscules) et qui renvoie le code alphabétique concaténé, le code additionné de `mot`, ainsi qu'un booléen qui indique si `mot` est parfait ou pas.

Correction

Sujet 22.2 : https://glassus.github.io/terminale_nsi/T6_6_Epreuve_pratique/BNS_2023/?s=03

```python
dico = { c:i+1 for i,c in enumerate("ABCDEFGHIJKLMNOPQRSTUVWXYZ") }

def est_parfait(mot):
    # mot est une chaîne de caractères (en lettres majuscules)
    code_concatene = ""
    code_additionne = 0
    for c in mot:
        code_concatene += str(dico[c])
        code_additionne += dico[c]
    code_concatene = int(code_concatene)
    if code_concatene % code_additionne == 0:
        mot_est_parfait = True
    else:
        mot_est_parfait = False
    return code_additionne, code_concatene, mot_est_parfait

for mot in ["PAUL","ALAIN"]:
    print(f"est_parfait({mot}) renvoie {est_parfait(mot)}")
```

```
est_parfait(PAUL) renvoie (50, 1612112, False)
est_parfait(ALAIN) renvoie (37, 1121914, True)
```

Exercice 6.95 (Comptage des doigts)

Une petite fille vient d'apprendre à compter de 1 à N en utilisant les cinq doigts de sa main gauche comme suit : elle commence par appeler son pouce 1, l'index 2, le majeur 3, l'annulaire 4 et l'auriculaire 5. Ensuite, elle inverse la direction, en appelant l'annulaire 6, le majeur 7, l'index 8 et son pouce 9, après quoi elle appelle son index 10, et ainsi de suite.

Supposons que nous indexions sa main gauche du pouce à l'auriculaire dans l'ordre des doigts 1 à 5. Si elle continue à compter comme décrit ci-dessus, sur quel doigt s'arrêtera-t-elle pour une valeur donnée de N ?

Implémentez une fonction nommée `Fingers` qui prend un entier N en entrée et renvoie le numéro du doigt sur lequel la petite fille s'arrêtera.

Par exemple, pour N = 8, la fonction doit renvoyer 2 ; pour N = 72 la fonction doit renvoyer 2.

Correction

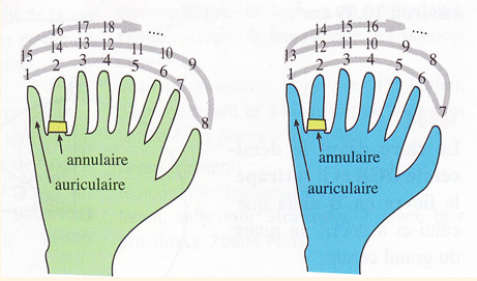
```
def fingers(N):
    cycle = 8
    index = (N - 1) % cycle + 1
    if index > 5:
        index = 10 - index
    return index

# TESTS
print(*[ f"fingers({N}) = {fingers(N)}" for N in (8, 10, 12, 72) ], sep=", ")
```

```
fingers(8) = 2, fingers(10) = 2, fingers(12) = 4, fingers(72) = 2
```

Exercice Bonus 6.96 (Défi Turing n° 19 – Rencontre du quatrième type)

Des petits hommes verts rencontrent des petits hommes bleus.

À leur grand étonnement, ils constatent que leurs mains ne comportent pas le même nombre de doigts : 7 pour les bleus et 8 pour les verts. Mais les savants des deux peuples remarquent que si l'on compte sur les doigts comme indiqué sur la figure, en faisant des allers-retours de l'auriculaire vers le pouce, puis du pouce vers l'auriculaire, certains nombres se comptent à la fois sur l'annulaire des mains bleues et sur celui des mains vertes (le 2 et le 14 par exemple). Ces nombres ont été qualifiés d'*annulaires* par les savants. Calculer la somme des nombres annulaires compris entre 1 et 2013.



Correction

Code *brute force*:

```
def nb_sur_annulaire(dm,nb_max):
    # dm = doigts de la main
    A = [2]
    i = 1
    while A[-1]<nb_max-(dm-2)*2:
        A.append(A[-1]+(dm-2)*2)
        A.append(A[-1]+2)
        i+=1
    return A

# TEST
nb_max = 25
V = nb_sur_annulaire(8,nb_max)
B = nb_sur_annulaire(7,nb_max)
Communs = [i for i in V if i in B]
print(f"Nombres sur l'annulaire de la main verte: {V}")
print(f"Nombres sur l'annulaire de la main bleu: {B}")
print(f"Nombres en commun {Communs}")

# DEFI
nb_max = 2013
V = nb_sur_annulaire(8,nb_max)
B = nb_sur_annulaire(7,nb_max)
s = sum([i for i in V if i in B])
print(s)
```

```
Nombres sur l'annulaire de la main verte: [2, 14, 16]
Nombres sur l'annulaire de la main bleu: [2, 12, 14, 24, 26]
Nombres en commun [2, 14]
94848
```

En prenant l'auriculaire comme point de départ on voit qu'il y a un cycle de $2 \times 7 - 2 = 12$ pour les bleus et $2 \times 8 - 2 = 14$ pour les verts. On voit que les nombres annulaires sont à +2 ou 0 modulo 12 pour les bleus ou modulo 14 pour les verts.

```python
def ann(x,dm1,dm2):
    m1 = 2*dm1-2
    m2 = 2*dm2-2
    if (x%m1==2 or x%m1==0) and (x%m2==2 or x%m2==0):
        return True
s = sum([x for x in range(1,2014) if ann(x,8,7)])
print(s)
```

```
94848
```

Exercice Bonus 6.97 (Pydéfi – Vif d'or)

Si Nicolas Flamel est célèbre pour sa pierre philosophale, Léonard de Pise, de 200 ans son aîné, a aussi laissé sa trace chez les sorciers, en créant le vif d'or. Bien que les mouvements du vif d'or semblent erratiques, celui-ci a en fait pour fonction de joindre des points de l'espace dans un ordre préétabli.

Précisément, la position du vif d'or est définie par trois coordonnées : abscisse, ordonnée et hauteur. À partir de sa position (x, y, z), le vif va alors se déplacer en ligne droite vers sa position suivante donnée par $(y, z, (x + y + z)\%n)$. Pour rappel, l'opération % donne le reste de la division entière, et n est ici une valeur positive, supérieure à 1, fixée à l'avance.

Le vif d'or entame toujours sa course en $(0, 0, 1)$. Supposons que n soit égal à 85, alors le vif d'or joindra successivement les positions suivantes

```
1.  (0, 0, 1)
2.  (0, 1, 1)
3.  (1, 1, 2)
4.  (1, 2, 4)
5.  (2, 4, 7)
6.  (4, 7, 13)
7.  (7, 13, 24)
8.  (13, 24, 44)
9.  (24, 44, 81)
10. (44, 81, 64)
11. (81, 64, 19)
...
```

Au bout d'un certain temps, le vif d'or reviendra à sa position d'origine et suivra donc à nouveau exactement le même parcours. Dans l'exemple précédent, cela se produit à la position 2977 :

```
...
2974. (0, 84, 1)
2975. (84, 1, 0)
2976. (1, 0, 0)
2977. (0, 0, 1)
2978. (0, 1, 1)
2979. (1, 1, 2)
...
```

En choisissant la valeur de n, on fait donc varier la trajectoire du vif d'or, ainsi que le temps qu'il met avant de revenir à sa position d'origine.

Léonard de Pise a calculé les 10 meilleures valeurs de n et les a entrées dans tous les vifs d'or. Il s'agit des valeurs de n, inférieures ou égales à 200, qui donnent les parcours les plus longs (c'est-à-dire pour lesquels le vif d'or effectue le plus de mouvements avant de repasser par sa position initiale).

Afin d'essayer de prévoir les déplacements du vif d'or lors du prochain match de Quidditch, vous avez en tête de découvrir ces 10 valeurs.

Source : https://pydefis.callicode.fr/defis/TrajetVifOr/txt

Exercice Bonus 6.98 (Défi Turing n° 52 – multiples constitués des mêmes chiffres)
On peut constater que le nombre 125874 et son double 251748 sont constitués des mêmes chiffres, mais dans un ordre différent. Trouver le plus petit $n \in \mathbb{N}^*$ tel que n, $2n$, $3n$, $4n$, $5n$ et $6n$ soient constitués des mêmes chiffres.

Correction

```
f = lambda n : sorted([int(c) for c in str(n)])

def Turing_52():
    n = 1
    B = f(n)
    while B!=f(2*n) or B!=f(3*n) or B!=f(4*n) or B!=f(5*n) or B!=f(6*n):
        n += 1
        B = f(n)
    return n

n = Turing_52()
print(f"n={n}, 2n={2*n}, 3n={3*n}, 4n={4*n}, 5n={5*n}, 6n={6*n}")
```

```
n=142857, 2n=285714, 3n=428571, 4n=571428, 5n=714285, 6n=857142
```

Exercice Bonus 6.99 (Défi Turing n° 61 – Non à l'isolement !)
Un chiffre est isolé s'il a comme voisin gauche et voisin droit un chiffre différent de lui-même. Par exemple, dans 776 444, 6 est isolé, mais les autres chiffres ne le sont pas.

D'autre part, les trois premiers nombres non multiples de 10 dont les carrés ne contiennent aucun chiffre isolé sont : $88^2 = 7744$, $74162^2 = 5500002244$ et $105462^2 = 11122233444$. Quel est le quatrième ?

Correction

```
def is_isolated(N):
    s = str(N)
    if s[0]!=s[1] or s[-1]!=s[-2]: # s[0] ou s[-1] est isolé
        return False
    for i in range(1,len(s)-1):
        if s[i-1]!=s[i] and s[i]!=s[i+1]: # s[i] est isolé
            return False
    return True

#print( is_isolated(87**2), is_isolated(88**2) )

L = []
n = 88
while len(L)<4:
    if is_isolated(n*n): L.append(n)
    n+=1
    if n%10==0: n+=1
print(L)
```

```
[88, 74162, 105462, 2973962]
```

Exercice 6.100 (Code César)
Le codage de César est une manière de crypter un message de manière simple : on choisit un nombre n (appelé clé de codage) et on décale toutes les lettres de notre message du nombre choisi. Exemple avec $n = 2$: la lettre "A" deviendra "C", le "B" deviendra "D" ... et le "Z" deviendra "B". Ainsi, le mot "MATHS" deviendra, une fois codé, "OCVJU" (pour décoder, il suffit d'appliquer le même algorithme avec $n = -2$). La question à laquelle il faut répondre a été codée : si la question est "TRGZKRCVWIRETV" et la clé de codage est $n = 17$, quelle est la réponse à la question ?

Correction

```python
def cesar(n,msg):
    abc = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    new = ""
    for lettre in msg:
        i = abc.index(lettre)
        new += abc[(i+n)%26]
    return new
```

```python
# TESTS
print(cesar(  2,'MATHS'))
print(cesar( -2,'OCVJU'))
print(cesar(-17,'TRGZKRCVWIRETV'))
```

```
OCVJU
MATHS
CAPITALEFRANCE
```

La réponse est donc Paris.

Remarque : pour connaître le code ASCII (entier) associé à un caractère on peut utiliser la fonction `ord()` :

```python
>>> ord("a")
97
>>> ord("z")
122
>>> ord("z")-ord("a")+1 # nb de caractères entre 'a' et 'z'
26
```

Réciproquement, pour connaître le caractère associé à un code ASCII on peut utiliser la fonction `chr()` :

```python
>>> chr(65)
'A'
>>> chr(90)
'Z'
>>> chr(ord("a")+6) # 6ième caractère après 'a'
'g'
```

On peut alors réécrire le code de César comme

```python
def cesar(n,msg):
    new = ""
    for lettre in msg:
        new += chr(((ord(lettre) - ord("A") + n) % 26) + ord("A"))
    return new

print(cesar(  2,'MATHS'))
print(cesar( -2,'OCVJU'))
print(cesar(-17,'TRGZKRCVWIRETV'))
```

```
OCVJU
MATHS
CAPITALEFRANCE
```

Exercice Bonus 6.101 (Défi Turing n° 17 – Nombres amicaux)

Pour un entier $n \in \mathbb{N}$ donné, on note $d(n)$ la somme des diviseurs propres de n (diviseurs de n inférieurs à n). On dit que deux entiers $a, b \in \mathbb{N}$ avec $a \neq b$ sont amicaux si $d(a) = b$ et $d(b) = a$. Par exemple,

$$a = 220 \text{ est divisible par } 1,2,4,5,10,11,20,22,44,55,110 \qquad b = 284 \text{ est divisible par } 1,2,4,71,142$$

$$d(a) = 1+2+4+5+10+11+20+22+44+55+110 = 284 = b \qquad d(b) = 1+2+4+71+142 = 220 = a$$

donc 220 et 284 sont amicaux.

Trouvez la somme de tous les nombres amicaux inférieurs à 10000.

Correction

Le plus simple est de tester, pour tous les nombres $n > 1$ si, lorsque $m = d(n)$, alors $n = d(m)$.

La fonction dp prend en entrée un entier $n \in \mathbb{N}$ et renvoie la somme des diviseurs propres de n.

```python
dp = lambda n : sum([x for x in range(1,int(n/2)+1) if (n%x==0)])

Amis = []
for n in range(1,10000+1):
    m=dp(n)
    if m!=n and dp(m)==n :
        Amis.append(n)
print(sum(Amis))
```

```
31626
```

Exercice Bonus 6.102 (The Longest Word)

Écrivez une fonction qui prend une chaîne de caractères sans signes de ponctuation en entrée, et retourne le mot le plus long dans la chaîne. Si plusieurs mots ont la même longueur, retournez le premier qui apparaît.

Exemple : si l'entrée est "this is a sentence" la sortie sera "sentence".

Source : https://py.checkio.org/en/mission/the-longest-word/

Correction

La fonction `longest_word` prend en entrée une chaîne de caractères `sentence`. Elle divise cette chaîne en une liste de mots en utilisant l'espace comme séparateur grâce à la méthode `split()`. Ensuite, elle utilise la fonction `max()` pour trouver l'élément de la liste L (les mots) qui a la longueur maximale en utilisant la fonction `len` comme clé de comparaison. Ainsi, elle retourne le mot le plus long. Si la liste est vide (ce qui peut arriver si la chaîne d'origine est vide), la valeur par défaut retournée est une chaîne vide, spécifiée avec l'argument `default=""`.

```python
def longest_word(sentence: str) -> str:
    L = sentence.split()
    return max(L,key=len,default="")

# Tests aléatoires
import random
import string

def generate_random_word(length):
    return ''.join(random.choice(string.ascii_lowercase) for _ in range(length))

def generate_random_sentence(word_count, max_word_length):
    words = [generate_random_word(random.randint(1, max_word_length)) for _ in range(word_count)]
    return ' '.join(words)

for _ in range(5):
    sentence = generate_random_sentence(random.randint(2, 10), 10)
    print(f"Sentence: {sentence}")
    result = longest_word(sentence)
    print(f"Longest word: {result}\n")
```

```
Sentence: fndwug lpjyfk ligdc uawltfjg fyyzlip
Longest word: uawltfjg

Sentence: rwtgyj svhomfiot
Longest word: svhomfiot

Sentence: oa ptxbmvknkj nage onwnxkhck mroptskl ax
Longest word: ptxbmvknkj

Sentence: ti ziysrok gbarekper mw fhbhwcriiw om c
Longest word: fhbhwcriiw

Sentence: fslyuvyoq dl d omwees fzuckafzq ay vubkkrse
Longest word: fslyuvyoq
```

Exercice Bonus 6.103 (Tickets t+ RATP)

Au moment où cet exercice est rédigé, 1 ticket de métro vaut 1.90 € mais si on les achète par carnet de dix, le prix du carnet est de 16.90 €.

On veut calculer le nombre de tickets et le nombre de carnet à acheter en minimisant la dépense (quitte à garder des tickets inutilisés)

Par exemple, avec 9 tickets il convient d'acheter un carnet et garder 2 tickets inutilisés plutôt qu'acheter 9 tickets à l'unité car sans carnet on paye $1.90 \times 9 = 17.10$ € tandis qu'un carnet coûte 16.90 €.

Écrire une script qui calcule la somme minimale à payer si on prévoit `n` voyages ; il devra aussi afficher combien de carnet acheter et s'il faut acheter des tickets à l'unité ou combien de voyages non utilisés restent sur un carnet.

Correction

```
prixTicket = 1.90
prixCarnet = 16.90

def RATP(n):
    somme_due = n*prixTicket
    nbCarnets = 0
    nbTickets = n
    if somme_due <= prixCarnet:
        print(f"Pour {n} voyage(s)")
        print(f"\ton achète {nbTickets} ticket(s) à l'unité et on paye {somme_due:.2f}€")
        return
    while somme_due > prixCarnet:
        nbCarnets += 1
        nbTickets -= 10
        somme_due -= prixCarnet
    nbTicketsRestes = max(0,10*nbCarnets-n)
    nbTicketsAAcheter = max(0,nbTickets)
    somme_due=nbCarnets*prixCarnet+nbTicketsAAcheter*prixTicket

    print(f"Pour {n} voyage(s)")
    print(f"\tsans carnet on payerait {n*prixTicket:.2f}")
    if nbTicketsRestes>0:
        print(f"\tsi on achète {nbCarnets} carnet(s) on paye {somme_due:.2f}€ et il reste
        ↪ {nbTicketsRestes} voyage(s)")
    else:
        if nbTicketsAAcheter>0:
            print(f"\tsi on achète {nbCarnets} carnet(s) et {nbTicketsAAcheter} ticket(s) à l'unité on
            ↪ paye {somme_due:.2f}€")
        else:
            print(f"\tsi on achète {nbCarnets} carnet(s) on paye {somme_due:.2f}€")
    return

# TEST
for n in [8,9,10,11,12,20,21]: # nombres de voyages
    RATP(n)
```

```
Pour 8 voyage(s)
→on achète 8 ticket(s) à l'unité et on paye 15.20€
Pour 9 voyage(s)
→sans carnet on payerait 17.10
→si on achète 1 carnet(s) on paye 16.90€ et il reste 1 voyage(s)
Pour 10 voyage(s)
→sans carnet on payerait 19.00
→si on achète 1 carnet(s) on paye 16.90€
Pour 11 voyage(s)
→sans carnet on payerait 20.90
→si on achète 1 carnet(s) et 1 ticket(s) à l'unité on paye 18.80€
Pour 12 voyage(s)
→sans carnet on payerait 22.80
→si on achète 1 carnet(s) et 2 ticket(s) à l'unité on paye 20.70€
Pour 20 voyage(s)
→sans carnet on payerait 38.00
→si on achète 2 carnet(s) on paye 33.80€
Pour 21 voyage(s)
→sans carnet on payerait 39.90
→si on achète 2 carnet(s) et 1 ticket(s) à l'unité on paye 35.70€
```

Variante:

```
def RATP(n):
    nbCarnets, nbTickets = divmod(n,10)
    if nbTickets*prixTicket>=prixCarnet :
        nbCarnets += 1
        nbTickets -= 10
```

```
    nbTicketsRestes = max(0,10*nbCarnets-n)
    nbTicketsAAcheter = max(0,nbTickets)
    somme_due = nbCarnets*prixCarnet+nbTicketsAAcheter*prixTicket
    if nbCarnets==0:
        print(f"Pour {n} voyage(s)")
        print(f"\ton achète {nbTickets} ticket(s) à l'unité et on paye {somme_due:.2f}€")
        return
    print(f"Pour {n} voyage(s)")
    print(f"\tsans carnet on payerait {n*prixTicket:.2f}")
    if nbTicketsRestes>0:
        print(f"\tsi on achète {nbCarnets} carnet(s) on paye {somme_due:.2f}€ et il reste
         ↪ {nbTicketsRestes} voyage(s)")
    else:
        if nbTicketsAAcheter>0:
            print(f"\tsi on achète {nbCarnets} carnet(s) et {nbTicketsAAcheter} ticket(s) à l'unité on
             ↪ paye {somme_due:.2f}€")
        else:
            print(f"\tsi on achète {nbCarnets} carnet(s) on paye {somme_due:.2f}€")
    return
```

Exercice Bonus 6.104 (Tickets réseau Mistral)

Écrire une script qui calcule la somme minimale à payer et le type de Tickets du réseau Mistral à acheter si on prévoit vBus voyages en bus et vBat voyages en bateau-bus. Au moment où cet exercice est rédigé voici les prix:

- 1 voyage terrestre coûte 1.40 €
- 1 voyage maritime coûte 2.00 €
- 1 carnet de 10 voyages terrestres ou maritimes coûte 10.00 €

Correction

```
cBus = 1.40
cBat = 2.00
cCar = 10.00

def Mistral(vBus,vBat):
    pBus = vBus*cBus
    pBat = vBat*cBat
    p = pBus+pBat
    print(f"\tsans carnet on paye {p:.2f}€")

    nCar = 0
    v = vBus+vBat
    while v>10 or p>cCar : # 10 car 1 carnet = 10 voyages
        nCar += 1
        v -= 10
        if vBat>=10:
            vBat -= 10
        else:
            vBus = vBus-(10-vBat)
            vBat = 0
        p = vBus*cBus+vBat*cBat

    return nCar, vBus, vBat, v, nCar*cCar+max(0,vBus)*cBus+max(0,vBat)*cBat

# TEST
for vBus,vBat in [(7,0),(8,0),(0,6),(5,6),(3,11),(3,13)]:
    print(f"Pour {vBus+vBat} voyage(s) dont {vBus} en bus et {vBat} en mer")
    nCar, vBus, vBat, v, p = Mistral(vBus,vBat)
    if nCar>0:
        if v<=0:
            print(f"\tavec {nCar} carnet(s), on paye {p:.2f}€ et il reste {-v} voyages")
        else:
            print(f"\tavec {nCar} carnet(s), {vBus} ticket(s) de bus et {vBat} de bateau, on paye
             ↪ {p:.2f}€")
```

```
Pour 7 voyage(s) dont 7 en bus et 0 en mer
→sans carnet on paye 9.80 €
Pour 8 voyage(s) dont 8 en bus et 0 en mer
→sans carnet on paye 11.20 €
→avec 1 carnet(s), on paye 10.00 € et il reste 2 voyages
Pour 6 voyage(s) dont 0 en bus et 6 en mer
→sans carnet on paye 12.00 €
→avec 1 carnet(s), on paye 10.00 € et il reste 4 voyages
Pour 11 voyage(s) dont 5 en bus et 6 en mer
→sans carnet on paye 19.00 €
→avec 1 carnet(s), 1 ticket(s) de bus et 0 de bateau, on paye 11.40 €
Pour 14 voyage(s) dont 3 en bus et 11 en mer
→sans carnet on paye 26.20 €
→avec 1 carnet(s), 3 ticket(s) de bus et 1 de bateau, on paye 16.20 €
Pour 16 voyage(s) dont 3 en bus et 13 en mer
→sans carnet on paye 30.20 €
→avec 2 carnet(s), on paye 20.00 € et il reste 4 voyages
```

Variante

```python
cBus = 1.40
cBat = 2.00
cCar = 10.00

def Mistral(vBus,vBat):
    print(f"\tsans carnet on paye {cBus*vBus+cBat*vBat:.2f} €")
    nbCarnets, r = divmod(vBus+vBat,10) # 10 car 1 carnet = 10 voyages
    if 10*nbCarnets >= vBat:
        vBat = 0
        vBus = r
        if cBus*vBus > cCar :
            nbCarnets += 1
            vBus -= 10
    else :
        vBat = r-vBus
        if cBat*vBat+cBus*vBus >= cCar :
            nbCarnets += 1
            vBus -= 10-vBat
            vBat = 0

    v = vBus+vBat
    p = nbCarnets*cCar+max(0,vBus)*cBus+max(0,vBat)*cBat
    return nbCarnets, vBus, vBat, v, p

# TEST
for vBus,vBat in [(7,0),(8,0),(0,6),(5,6),(3,11),(3,13)]:
    print(f"Pour {vBus+vBat} voyage(s) dont {vBus} en bus et {vBat} en mer")
    nCar, vBus, vBat, v, p = Mistral(vBus,vBat)
    if nCar>0:
        if v<=0:
            print(f"\tavec {nCar} carnet(s), on paye {p:.2f} € et il reste {-v} voyages")
        else:
            print(f"\tavec {nCar} carnet(s), {vBus} ticket(s) de bus et {vBat} de bateau, on paye
            ↪ {p:.2f} € ")
```

Exercice Bonus 6.105 (Problème d'Euler n°9)

Le triplet d'entiers naturels non nuls (a,b,c) est pythagoricien si $a^2+b^2=c^2$. Par exemple, $(3,4,5)$ est un triplet pythagoricien car $3^2+4^2=9+16=25=5^2$. Trouver le seul triplet Pythagoricien pour lequel $a+b+c=1000$.

Correction

Nous savons que $a<c$, $b<c$ et nous pouvons étudier seulement les cas $a<b$. Nous n'avons pas besoin de faire varier les 3 valeurs, en faire varier 2 suffit car si nous connaissons, par exemple, la valeur de b et a, nous pouvons en déduire celle de c en résolvant l'équation $a+b+c=p$ où p est le périmètre (ici $p=1000$).

L'intérêt d'utiliser une fonction est de pouvoir quitter le calcul dès que le triplet a été trouvé (utiliser une instruction `break` ne permet de sortir que de la boucle la plus interne) :

```
>>> def Pytagore(p):
...     for a in range(1,int(p/3)+1): # au lieu de range(1,p-1) car a<b<c implique 3a<a+b+c=p
...         for c in range(int(p/3),p-a): # au lieu de range(a,p-a) car a<b<c implique 3c>a+b+c=p
...             b = p-a-c
...             if (a*a+b*b)==(c*c):
...                 return a,b,c
...
>>> print(Pytagore(1000))
(200, 375, 425)
```

Exercice Bonus 6.106 (PyDéfis – Cerbère)

Histoire : pour son douzième et dernier travail, Eurysthée tenta de se débarasser d'Hercule en lui demandant de ramener Cerbère, le molosse à trois têtes qui gardait la porte des enfers. Pour Hercule, la première étape était de commencer à naviguer sur le Styx, au bon endroit, afin d'être guidé jusqu'au trône d'Hadès.

Hermès lui indiqua tout d'abord à quelle distance exacte, en kilomètres, était situé le point d'embarquement sur le Styx. La recherche d'Hercule se limitait donc à un grand cercle autour de Mycènes. Puis, Athéna ajouta une précision d'importance :

> Si tu marches vers l'ouest, pendant un nombre entier de kilomètres, puis vers le nord, pendant un nombre entier de kilomètres, alors, tu arriveras précisément au point d'embarquement sur le Styx. Si plusieurs choix s'offrent à toi, choisis celui qui t'emmeneras le plus au nord.

Défi : aide Hercule en lui indiquant la position de l'entrée du Styx. Pour cela, il suffit de donner les deux valeurs : nombre de kilomètres à l'ouest, nombre de kilomètres au nord.

Testez votre code : si Hermès avait initialement indiqué à Hercule que la porte des enfers était située à 50 kilomètres, alors Hercule aurait pu déterminer sa position exacte. En effet, il y a quatre triangles rectangles avec des côtés entiers qui ont un grand côté (hypothénuse) de longeur 50 : les couples (base, hauteur) $(14,48)$, $(48,14)$, $(30,40)$, $(40,30)$. En choisissant le point qui mène le plus au nord, Hercule aurait su que la porte des enfers était située à 14 km à l'ouest et 48 km au nord. Pour aider Hercule, il aurait donc suffit de répondre $(14,48)$.

Source : https://pydefis.callicode.fr/defis/Herculito12Cerbere/txt

Exercice 6.107 (Palindromes)

On dit qu'un entier $n \geq 0$ est un palindrome s'il se lit à l'identique de gauche à droite ou de droite à gauche. Par exemple $n = 178496694871$ est un palindrome.

1. Écrire une fonction `isPal(n)` prenant en argument un entier n et répondant le booléen `True` ou `False` suivant que n est (ou n'est pas) un nombre palindrome.
2. Écrire une liste en compréhension qui contient les palindromes de l'intervalle $[100,300]$ (en utilisant la fonction précédente).
3. Écrire une liste en compréhension qui contient les nombre $n < 10^6$ non palindromes mais dont le carré est palindromique. Par exemple 836 en fait partie car $836^2 = 698896$ est palindromique.
4. Trouver le plus petit n non palindrome dont le cube est palindromique.

Correction

1. ```
   isPal = lambda n : str(n)==str(n)[::-1]
   ```
2. ```
   print( [ i for i in range(100,300+1) if isPal(i) ] )
   ```
   ```
   [101, 111, 121, 131, 141, 151, 161, 171, 181, 191, 202, 212, 222, 232, 242, 252, 262, 272, 282,
   ↪ 292]
   ```
3. ```
   L = [ i for i in range(1,10**6+1) if (isPal(i**2) and not isPal(i))  ]
   print(L)
   print(f"En effet, les carrés sont {[ell**2 for ell in L]}")
   ```
   ```
   [26, 264, 307, 836, 2285, 2636, 22865, 24846, 30693, 798644]
   En effet, les carrés sont [676, 69696, 94249, 698896, 5221225, 6948496, 522808225, 617323716,
   ↪ 942060249, 637832238736]
   ```
4. ```
   def cubPal():
       n = 1
   ```

```
    while not ( isPal(n**3) and not isPal(n) ) :
        n += 1
    return n

n = cubPal()
print(f"Plus petit entier non palindrome dont le cube est palindrome: {n = }, {n**3 = }")
Plus petit entier non palindrome dont le cube est palindrome: n = 2201, n**3 = 10662526601
```

Exercice Bonus 6.108 (Défi Turing n°4 – nombre palindrome)

Un nombre palindrome se lit de la même façon de gauche à droite et de droite à gauche. Le plus grand palindrome que l'on peut obtenir en multipliant deux nombres de deux chiffres est $9009 = 99 \times 91$. Quel est le plus grand palindrome que l'on peut obtenir en multipliant un nombre de 4 chiffres avec un nombre de 3 chiffres ?

Correction

Il s'agit de trouver le plus grand palindrome (nombre qui se lit de la même façon de gauche à droite que de droite à gauche) issu du produit de 2 nombres, l'un à 3 chiffres l'autre à 4. Nous pouvons donc résoudre ce problème de deux façons différentes :

- soit faire le produit de tous les nombres à 3 chiffres avec tous les nombres à 4 chiffres et regarder quel est le plus grand palindrome formé,
- soit lister tous les palindromes inférieurs à $999 \times 9999 = 9989001$ dans l'ordre décroissant et regarder le premier de cette liste qui est égal au produit de 2 nombres l'un à 3 chiffres l'autre à 4.

1-ère piste :

```
L = [a*b for a in range(100,1000) for b in range(1000,10000) if a*b==int(str(a*b)[::-1])]
print(max(L))
9744479
```

2-nde piste (plus rapide) : on utilise une fonction ainsi on pourra quitter le calcul dès qu'elle trouve le triplet (utiliser une instruction `break` ne permet de sortir que de la boucle la plus interne).

```
def cherche():
        for p in range(1000*10000,100*1000,-1):
                rev = int(str(p)[::-1])
                if p==rev:
                        for a in range(100,1000):
                                b,r1 = divmod(p,a)
                                if r1==0 and 1000<=b<10000:
                                        return(p,a,b)

p,a,b = cherche()
print(f"Le plus grand palindrome est {p} = {a} x {b}")
Le plus grand palindrome est 9744479 = 983 x 9913
```

Exercice Bonus 6.109 (PyDéfis – Le jour de la serviette)

Le 25 mai prochain, nous célébrerons la *Journée de la Serviette*. Pour marquer cette occasion spéciale, vous avez décidé de confectionner une serviette brodée d'une énigme mathématique. L'objectif est de trouver trois nombres entiers, tous inférieurs à 1000, dont la somme et le produit ne s'écrivent qu'avec les chiffres 4 et 2. Il existe de nombreux triplets répondant à ces critères. Par exemple, le triplet (18, 74, 332) donne une somme de 424 et un produit de 442224, utilisant uniquement les chiffres 2 et 4. Parmi tous les triplets possibles, quel est celui dont les trois nombres de départ contiennent le plus grand nombre de fois le chiffre 4 ? (Dans l'exemple précédent, seul le nombre 74 contient un 4.)

Source : https://pydefis.callicode.fr/defis/C25_4_42/txt

Exercice Bonus 6.110 (Pydéfi – Les bœufs de Géryon)

Histoire : pour son dixième travail, Eurysthée demanda à Hercule de lui rapporter les bœufs de Géryon. Celui-ci, un géant à trois torses, possédait un troupeau de magnifiques bœufs. Naturellement, il veillait jalousement sur son troupeau, aidé par Orthros, son chien tricéphale, et Eurythion, son dragon à sept têtes. Hercule n'eut pas à user de force pour s'emparer du troupeau, car Géryon était joueur et sous-estima les facultés de calcul d'Hercule. Géryon

proposa le marché suivant :

> Comme tu vois, j'ai trois sortes de bœufs : des blancs, qui sont les moins nombreux ; des roux ; des noirs, qui sont les plus nombreux. Et comme j'aime beaucoup les nombres, j'ai fait en sorte que si on multiplie le nombre de bœufs roux par le nombre de bœufs blancs et enfin par le nombre de bœufs noirs, le nombre obtenu est 777 fois plus grand que le nombre total de bœufs. Si tu m'indiques combien j'ai de bœufs en tout, tu pourras partir avec mon troupeau.

Hercule réfléchit un instant et annonça :

> Tu as 21 bœufs blancs, 74 roux et 95 noirs. En effet : $21 \times 74 \times 95 = 147630$ et ton troupeau compte $21 + 74 + 95 = 190$ animaux. Et nous avons bien $190 \times 777 = 147630$. Ma réponse est donc 190. Laisse-moi partir maintenant.

Géryon répondit :

> Tu calcules vite, Hercule, mais tu observes mal et ta réponse n'est pas correcte. Tu vois bien que dans mon troupeau, en réalité, le nombre de bœufs noirs vaut moins que le double du nombre de bœufs blancs. Et mon troupeau compte moins de 1000 têtes... Je te laisse une dernière chance...

Défi : pourrez-vous aider Hercule en lui soufflant le nombre de bœufs du troupeau ?
Source : https://pydefis.callicode.fr/defis/Herculito10Boeufs

Exercice Bonus 6.111 (Pydéfi – Le lion de Némée)

Histoire : le premier travail qu'Eurysthée demanda à Hercule fut de lui ramener la peau du lion de Némée. Le terrible animal vivait dans la forêt d'Argolide et terrorisait les habitants de la région. Hercule traversa donc la forêt d'Argolide à la recherche du lion. Là, il vit que des petits panneaux avaient été fixés sur certains arbres. Sur chaque panneau, le nom d'une divinité était inscrit. Hercule nota tous les noms qu'il rencontra. Approchant de l'antre du lion, il vit, gravé dans la pierre :

> La lettre "A" vaut 1, la lettre "B" vaut 2, jusqu'à la lettre "Z" qui vaut 26. Ainsi, le mot : "BABA" vaut 6 (=2+1+2+1). Cherche la valeur de chaque mot, classe-les de la plus petite valeur à la plus grande, et prononce les mots dans cet ordre : le lion se rendra à toi.

Hercule considéra sa liste de divinités :

```
ARTEMIS ASCLEPIOS ATHENA ATLAS CHARON CHIRON CRONOS DEMETER EOS ERIS EROS GAIA HADES HECATE
↪ HEPHAISTOS HERA HERMES HESTIA HYGIE LETO MAIA METIS MNEMOSYNE NYX OCEANOS OURANOS PAN
↪ PERSEPHONE POSEIDON RHADAMANTHE SELENE THEMIS THETIS TRITON ZEUS
```

Voyons : ARTEMIS vaut 85, donc il faut la placer avant ASCLEPIOS qui vaut 99...
Défi : pouvez-vous aider Hercule, en lui indiquant dans quel ordre il doit citer les divinités ?
Source : https://pydefis.callicode.fr/defis/Herculito01Lion

Exercice 6.112 (Conversion base 2 → base 10)

On modélise la représentation binaire d'un entier non signé par un tableau d'entiers dont les éléments sont 0 ou 1. Par exemple, le tableau [1, 0, 1, 0, 0, 1, 1] représente l'écriture binaire de l'entier dont l'écriture décimale est $1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 83$:

L_i	1	0	1	0	0	1	1
2^{6-i}	2^6	2^5	2^4	2^3	2^2	2^1	2^0
$L_i \times 2^{6-i}$	64	0	16	0	0	2	1

$\rightsquigarrow \sum_{i=0}^{6} L_i \times 2^{6-i} = 83$

Compléter la fonction `convertir2to10(L)` qui prend une liste d'entiers dont les éléments sont 0 ou 1 et représentant un entier écrit en binaire. Elle renvoie l'entier (dans l'écriture décimale usuelle) lui associé.
Exemple : `convertir2to10( [1, 0, 1, 0, 0, 1, 1] )` renvoie l'entier 83.

Correction

```
convertir2to10 = lambda L : sum([ v*2**i for i,v in enumerate(L[::-1])])
# TESTS
from random import randint
TESTS = []
TESTS.append( [1, 0, 1, 0, 0, 1, 1] )
TESTS.append( [1] )
```

```
TESTS.append( [1, 0] )
TESTS.append( [1, 1] )
TESTS.append( [ randint(0,1) for _ in range(randint(3,10))] )
TESTS.append( [ randint(0,1) for _ in range(randint(3,10))] )
TESTS.append( [ randint(0,1) for _ in range(randint(3,10))] )
TESTS.append( [ randint(0,1) for _ in range(randint(3,10))] )
for L in TESTS:
    print(f"convertir2to10({L}) = {convertir2to10(L)}")
```

```
convertir2to10([1, 0, 1, 0, 0, 1, 1]) = 83
convertir2to10([1]) = 1
convertir2to10([1, 0]) = 2
convertir2to10([1, 1]) = 3
convertir2to10([1, 1, 1, 1]) = 15
convertir2to10([1, 0, 1]) = 5
convertir2to10([1, 1, 0, 1, 0]) = 26
convertir2to10([0, 1, 0, 0, 0, 1, 0]) = 34
```

Exercice 6.113 (Conversion base 3 → base 10)

On modélise la représentation en base 3 d'un entier non signé par un tableau d'entiers dont les éléments sont 0 ou 1 ou 2. Par exemple, le tableau [2, 0, 1, 0, 0, 1, 2] représente l'écriture en base 3 de l'entier dont l'écriture décimale est $2\times3^6+0\times3^5+1\times3^4+0\times3^3+0\times3^2+1\times3^1+2\times3^0=1544$:

L_i	2	0	1	0	0	1	2	
3^{6-i}	3^6	3^5	3^4	3^3	3^2	3^1	3^0	
$L_i\times3^{6-i}$	1458	0	81	0	0	3	2	$\rightsquigarrow \sum_{i=0}^{6} L_i\times3^{6-i}=1544$

Compléter la fonction `convertir3to10(L)` qui prend une liste d'entiers dont les éléments sont 0 ou 1 ou 2 et représentant un entier écrit en base 3. Elle renvoie l'entier (dans l'écriture décimale usuelle) lui associé.

Exemple : `convertir3to10( [2, 0, 1, 0, 0, 1, 2] )` renvoie l'entier 1544.

Correction

```
convertir3to10 = lambda L : sum([ v*3**i for i,v in enumerate(L[::-1])])
# TESTS
from random import randint
TESTS = []
TESTS.append( [2, 0, 1, 0, 0, 1, 2] )
TESTS.append( [1] )
TESTS.append( [1, 0] )
TESTS.append( [1, 1] )
TESTS.append( [ randint(0,1) for _ in range(randint(3,10))] )
TESTS.append( [ randint(0,1) for _ in range(randint(3,10))] )
TESTS.append( [ randint(0,1) for _ in range(randint(3,10))] )
TESTS.append( [ randint(0,1) for _ in range(randint(3,10))] )
for L in TESTS:
    print(f"convertir3to10({L}) = {convertir3to10(L)}")
```

```
convertir3to10([2, 0, 1, 0, 0, 1, 2]) = 1544
convertir3to10([1]) = 1
convertir3to10([1, 0]) = 3
convertir3to10([1, 1]) = 4
convertir3to10([1, 1, 1, 0, 1, 0, 1]) = 1063
convertir3to10([0, 1, 1]) = 4
convertir3to10([0, 1, 0, 0, 1, 1, 1, 0, 1]) = 2305
convertir3to10([0, 1, 1, 1]) = 13
```

Exercice 6.114 (Écriture d'un entier dans une base quelconque et entiers brésiliens)

1. On considère un entier n écrit en base 10. Écrire une fonction qui renvoie ses chiffres dans son écriture dans une base b quelconque : c'est la liste des restes obtenus dans l'ordre inverse.

   Exemple : 11 en base 10 (*i.e.* $1\times10^1+1\times10^0$) s'écrit $\underline{1011}_2$ en base 2 car $1\times2^3+0\times2^2+1\times2^1+1\times2^0=8+0+2+1$.

   http://villemin.gerard.free.fr/Wwwgvmm/Numerati/Base.htm

2. On dit qu'un entier n est brésilien en base b si tous ses chiffres sont égaux dans cette base de numération. On suppose $1 < b < n-1$ (car on a toujours $n = \underline{11}_{n-1}$ en base $b = n-1$ ainsi, si $b \geq n$, l'entier n se réduirait au seul chiffre n). Montrer que l'entier 80 est cinq fois brésilien (c'est-à-dire dans cinq bases de numération différentes) en utilisant la fonction précédente.
   Un article sur les nombres brésiliens https://oeis.org/A125134/a125134.pdf

Correction
Changement de base :

```
def base(n,b):
    L = []
    q,r = divmod(n,b)
    L.append(r)
    while q>0:
        q,r = divmod(q,b)
        L.append(r)
    return L[::-1]

print(base(11,2))
```

```
[1, 0, 1, 1]
```

80 est brésiliens dans les bases suivantes :

```
def bres80():
    B = []
    for b in range(2,80):
        ecriture = base(80,b)

        s = ""
        for c in ecriture:
            s += str(c)

        if len(set(ecriture))==1:
            B.append( (b,f"car il s'écrit {s}") )
            if len(B)==5:
                return B

print(f"80 est brésilien dans les bases suivantes: {bres80()}")
```

```
80 est brésilien dans les bases suivantes: [(3, "car il s'écrit 2222"), (9, "car il s'écrit 88"), (15,
↪  "car il s'écrit 55"), (19, "car il s'écrit 44"), (39, "car il s'écrit 22")]
```

Exercice 6.115 (Spreadsheet Column Number)

Les feuilles de calcul utilisent un système de nommage alphabétique pour les colonnes, commençant par 'A', 'B', 'C', etc. Les noms de colonnes continuent comme 'AB', etc. Étant donné une chaîne représentant un titre de colonne dans une feuille de calcul, déterminez son numéro de colonne correspondant. Dans les feuilles de calcul, les colonnes sont étiquetées en utilisant un système de base 26 avec les 'chiffres' A-Z, où 'A' représente 1 et 'Z' représente 26. Ensuite, les colonnes à deux lettres commencent par 'AA' comme 27 et continuent jusqu'à 'ZZ' comme 702, et ainsi de suite.

Exemples :

- `column_number('A')` retourne 1 : Dans une feuille de calcul, la première colonne est étiquetée 'A', ce qui correspond à 1 dans le système de numérotation.
- `column_number('Z')` retourne 26 : La dernière colonne à une lettre dans une feuille de calcul est 'Z', ce qui correspond à 26.
- `column_number('AB')` retourne 28 : Cela représente une colonne à deux lettres. 'A' correspond à 1 et 'B' correspond à 2. Donc, c'est 26 (A) + 2 (B) = 28.
- `column_number('AZ')` retourne 52 : 'A' correspond à 1 et 'Z' correspond à 26. Donc, c'est 26 (A) + 26 (Z) = 52.
- `column_number('BA')` retourne 53 : La première lettre 'B' correspond à 2, et la seconde lettre 'A' correspond à 1. Donc, c'est 26 * 2 (B) + 1 (A) = 53.

- `column_number('BZ')` retourne 78 : La première lettre 'B' correspond à 2, et la seconde lettre 'Z' correspond à 26. Donc, c'est 26 * 2 (B) + 26 (Z) = 78.
- `column_number('ZY')` retourne 701 : La première lettre 'Z' correspond à 26, et la seconde lettre 'Y' correspond à 25. Donc, c'est 26 * 26 (Z) + 25 (Y) = 701.

Correction
Source : https://py.checkio.org/fr/mission/excel-column-number/

Un Spreadsheet utilise un système de base 26, avec les lettres A à Z représentant les chiffres de 1 à 26. C'est un peu comme un système décimal (base 10), mais avec des lettres à la place des chiffres.

Par exemple :

$$\texttt{"A"} \to 1, \qquad \texttt{"Z"} \to 26,$$
$$\texttt{"AA"} = 26^1 \cdot 1 + 26^0 \cdot 1 = 27, \qquad \texttt{"AB"} = 26^1 \cdot 1 + 26^0 \cdot 2 = 28.$$

```python
def column_number(name: str) -> int:
    return sum([ (ord(c)-64)*26**(len(name)-1-i) for i,c in enumerate(name)])

# import numpy as np
# def column_number(name: str) -> int:
#     return np.polyval([ord(c) - ord('@') for c in name], 26)

print(column_number("AA")) # 27
print(column_number("A"))  # 1
print(column_number("Z"))  # 26
print(column_number("AB")) # 28
print(column_number("ZY")) # 701
```

```
27
1
26
28
701
```

- La notation `ord('@')` donne 64, donc `ord('A') - 64 = 1`, `ord('B') - 64 = 2`, ..., `ord('Z') - 64 = 26`. Ainsi `ord(c) - 64` convertit une lettre majuscule en son "chiffre"-Spreadsheet. Exemple : "AB" devient [1, 2]. Puis on multiplie chaque nombre par la puissance de 26 correspondant à sa position (de gauche à droite). Dans notre exemple $1 \times 26^1 + 2 \times 26^0 = 26 + 2 = 28$.
- Si on utilise `NumPy`, on peut rendre cette fonction plus concise en évaluant directement un "polynôme en base 26". `np.polyval` évalue un polynôme dont les coefficients sont les valeurs des lettres, interprétées en base 26.

Exercice Bonus 6.116 (Suites de Kaprekar))

Soit $n \in \mathbb{N}$. Soit d (resp. c) l'entier formé des chiffres dans l'ordre décroissant (resp. croissant) de u. On pose $\mathrm{K}(u) = d - c$. On va construire une suite définie par récurrence comme suit :

$$\begin{cases} u_0 \\ u_{n+1} = \mathrm{K}(u_n) \end{cases}$$

Écrire une fonction qui calcule la suite.

Il est possible de montrer qu'il existe un indice p et un indice $m > p$ tel que la suite boucle sur la sous-liste $[u_p, u_{p+1}, \ldots, u_{m-1}]$ (appelée cycle de u_0 de période $m - p$).

En particulier, si $m = p + 1$, on a $u_m = u_p$ pour tout $m \geq p$ (la suite est stationnaire à partir de p, la période est donc $m - p = 1$).

1. Vérifier que, si $u_0 = 78545$, alors on a un cycle de période 4 (la sous-liste est $[82962, 75933, 63954, 61974]$).
2. Vérifier que, si $u_0 = 1100$, la suite est stationnaire à partir de $n = 4$ de valeur 6174.
3. Vérifier que, si u_0 a exactement 3 chiffres, la suite est stationnaire avec $u_p = 495$ ou 0. Quel est la valeur maximale de p ?
4. Vérifier que, si u_0 a exactement 4 chiffres, la suite est stationnaire avec $u_p = 6174$ ou 0. Quel est la valeur

maximale de p ?
https://fr.wikipedia.org/wiki/Algorithme_de_Kaprekar

Correction

```
def K(n):
    c = sorted(list(str(n)))
    d = c[::-1]
    return int(''.join(d))-int(''.join(c))

def suite(u0):
    uu = [u0]
    while True :
        uu.append(K(uu[-1]))
        if uu[-1] in uu[:-1]:
            p = uu.index(uu[-1])
            m = len(uu)-1
            return p,m,uu[p:-1],u0
```

Base 10

```
t = suite(78545)
sol = f"u_0={t[-1]}, p={t[0]}, m={t[1]}, periode={len(t[2])}, cycle={t[2]}"
print(sol)

t = suite(7641)
sol = f"u_0={t[-1]}, p={t[0]}, m={t[1]}, periode={t[1]-t[0]}, cycle={t[2]}"
print(sol)
t = suite(495)
sol = f"u_0={t[-1]}, p={t[0]}, m={t[1]}, periode={len(t[2])}, cycle={t[2]}"
print(sol)

t = suite(1100)
sol = f"u_0={t[-1]}, p={t[0]}, m={t[1]}, periode={t[1]-t[0]}, cycle={t[2]}"
print(sol)
```

```
u_0=78545, p=2, m=6, periode=4, cycle=[82962, 75933, 63954, 61974]
u_0=7641, p=1, m=2, periode=1, cycle=[6174]
u_0=495, p=0, m=1, periode=1, cycle=[495]
u_0=1100, p=4, m=5, periode=1, cycle=[6174]
```

u_0 à trois chiffres :

```
T = []
for n in range(100,1000):
    t = suite(n)
    if len(t[2])>1:
        print("Non constante")
        break
    T.append(t)

print(set([t[2][0] for t in T ]))
```

```
{0, 495}
```

u_0 à quatre chiffres :

```
T = []
for n in range(1000,10000):
    t = suite(n)
    if len(t[2])>1:
        print("Non constante")
        break
    T.append(t)

print(set([t[2][0] for t in T ]))
```

```
{0, 6174}
```

Exercice Bonus 6.117 (Pydéfi – Numération Gibi : le Neutubykw)

Les Gibis sont plus calés en histoire des sciences que les Shadoks. Ils ont leur système de numération spécial, en base 4, mais leurs chiffres sont une référence à des personnages célèbres de l'histoire de l'informatique :

KWA pour le 0 en l'honneur d'Al Khwarizmi

BY pour le 1, rappelant Ada Byron (plus connue sous le nom d'Ada Lovelace)

TU pour le 2, afin que personne n'oublie Alan Turing

NEU pour le 3, nous rappelant combien John von Neumann était brillant

Cette numération s'appelle le Neutubykwa. On trouve l'écriture d'un nombre en Neutubykwa en divisant la quantité à écrire en paquets de 4.

Pour écrire 118, par exemple, on réalise 29 paquets de 4, et il reste 2 unités (118=29*4+2). TU (2) est donc le chiffre des unités. Pour avoir le chiffre des quatrains (celui qui est juste à gauche des unités), on prend les 29 paquets qu'on regroupe par groupes de paquets de 4. Cela donne 7 groupes de paquets de 4, et il restera un seul paquet de 4. Le chiffre des quatrains est donc BY (1). Pour le chiffre situé à gauche de BY, on recommence : les 7 paquets sont groupés en 1 seul tas de 4 et il reste 3 paquets. Le chiffre suivant est donc NEU (3). Et enfin, le tas restant est groupé en 0 paquets de 4, et il reste 1 tas. Le premier chiffre du nombre est donc BY. Ainsi, 118, s'écrit BYNEUBYTU.

Testez votre code : si la liste d'entrée était [118, 3, 24], il faudrait répondre `'BYNEUBYTU'`, `'NEU'`, `'BYTUKWA'`.

Défi : donnant leur équivalent en Neutubykwa des nombres [363, 204, 108, 181, 326, 154, 259, 289, 332, 137].

Source : https://pydefis.callicode.fr/defis/Neutubykwa

Exercice Bonus 6.118 (Pydéfi – Pokédex en vrac)

Dans l'univers Pokémon, un dresseur est une personne qui capture des Pokémons sauvages, les élève et les entraîne à combattre les Pokémons d'autres dresseurs. Le protagoniste de chaque version des jeux vidéo Pokémon est un dresseur ambitieux ; Sacha est le plus célèbre d'entre eux.

Sacha a déjà capturé de nombreux Pokémons et son Pokédex est presque rempli ! Mais il n'est pas très ordonné, il a juste noté au fur et à mesure les numéros des Pokémons qu'il a capturés... Il vous confie son Pokédex et vous demande de lui fournir une vue synthétique de sa liste de Pokémons :

- les numéros des Pokémons sont listés par séquences séparées par des virgules,
- les séquences peuvent être de la forme :
  - $i - j$ indiquant que le Pokédex contient les Pokémons du numéro i au numéro j inclus, mais pas les Pokémons $i-1$ et $j+1$,
  - k indiquant que le Pokédex contient le Pokémon de numéro k, mais pas les Pokémons $k-1$ et $k+1$
  - les séquences sont ordonnées de manière croissante.

Par exemple si le Pokédex contenait les numéros suivants `2, 10, 5, 1, 7, 9, 8` la vue synthétique serait représentée par `1-2,5,7-10`.

Le Pokédex de Sacha est bien plus vaste, vous pouvez le télécharger ici : https://pydefis.callicode.fr/defis/C22_GenList/input. Saurez-vous lui indiquer sa vue synthétique ?

https://pydefis.callicode.fr/defis/C22_GenList/txt

Exercice Bonus 6.119 (Défi Turing n°141 – Combien de 6 ?)

On multiplie les chiffres composant un nombre plus grand que 9. Si le résultat est un nombre à un chiffre, on l'appelle l'image du nombre de départ. S'il a plus d'un chiffre, on répète l'opération jusqu'à obtenir un nombre à un chiffre. Par exemple

$$666 \xrightarrow[6\times6\times6]{} 216 \xrightarrow[2\times1\times6]{} 12 \xrightarrow[1\times2]{} 2$$

Combien de nombres entre 10 et 10 millions ont comme image 6 ?

Correction

```
# def prod(L):
# ⟶p = 1
# ⟶for ell in L:
# ⟶⟶p *= ell
# ⟶return p

# depuis python 3.8 on peut écrire
```

```
from math import prod

d = { i:0 for i in range(10) }
for i in range(10,10**7+1):
    while len(str(i))>1:
        i = prod([int(i) for i in str(i)])
    d[i] += 1
print(d)
```

```
{0: 9394518, 1: 6, 2: 86092, 3: 27, 4: 6173, 5: 7413, 6: 318456, 7: 27, 8: 187196, 9: 83}
```

Exercice Bonus 6.120 (Défi Turing n°33 – Pâques en avril)

Durant les années 2001 à 9999 (bornes comprises), combien de fois la date de Pâques tombera-t-elle en avril, dans le calendrier grégorien ?

Cf. https://fr.wikipedia.org/wiki/Calcul_de_la_date_de_P%C3%A2ques_selon_la_m%C3%A9thode_de_Gauss

Correction

```
def paques(a):
    n = a%19 # cycle de Méton
    c,u = divmod(a,100) # centaine et rang de l'année
    s,t = divmod(c,4) # siècle bissextile
    p = (c+8)//25 # cycle de proemptose
    q = (c-p+1)//3 # proemptose
    e = (19*n+c-s-q+15)%30 # épacte
    b,d = divmod(u,4) # année bissextile
    L = (2*t+2*b-e-d+32)%7 # lettre dominicale
    h = (n+11*e+22*L)//451 # correction
    m,j = divmod(e+L-7*h+114,31)
    # print(f'{a = }, {n = }, {c = }, {u = }, {s = }, {t = }, {p = }, {q = }, {e = }, {b = }, {d = }, {L
    ↪  = }, {h = }, {m = }, {j = } ')
    return m,j

def afficher(a):
    x = paques(a)
    m = x[0]
    j = x[1]
    if m==3:
        print(f"En {a} le dimanche de Pâques est le {j+1} mars")
    elif m==4:
        print(f"En {a} le dimanche de Pâques est le {j+1} avril")
    else:
        print("Error")
    
# TEST
for a in range(2020,2031):
    afficher(a)

# # DEFIS
# dico = {3:0,4:0}
# for a in range(2001,9999+1):
# ⟶m,_ = paques(a)
# ⟶dico[m] += 1
#
# print(dico)
```

```
En 2020 le dimanche de Pâques est le 12 avril
En 2021 le dimanche de Pâques est le 4 avril
En 2022 le dimanche de Pâques est le 17 avril
En 2023 le dimanche de Pâques est le 9 avril
En 2024 le dimanche de Pâques est le 31 mars
En 2025 le dimanche de Pâques est le 20 avril
En 2026 le dimanche de Pâques est le 5 avril
```

```
En 2027 le dimanche de Pâques est le 28 mars
En 2028 le dimanche de Pâques est le 16 avril
En 2029 le dimanche de Pâques est le 1 avril
En 2030 le dimanche de Pâques est le 21 avril
```

Exercice Bonus 6.121 (Sun angle)

Tout vrai voyageur doit savoir faire 3 choses : réparer le feu, trouver l'eau et extraire des informations utiles de la nature qui l'entoure. La programmation ne vous aidera pas avec le feu et l'eau, mais en ce qui concerne l'extraction d'informations, c'est peut-être exactement ce dont vous avez besoin. Votre tâche est de trouver l'angle du soleil au-dessus de l'horizon en connaissant l'heure de la journée.

On suppose que le soleil se lève à l'Est à 6h00, ce qui correspond à l'angle de 0° ; à 12h00, le soleil atteint son zénith, ce qui signifie que l'angle est égal à 90° ; 18h00 est l'heure du coucher du soleil, l'angle est donc de 180°.

Écrire une fonction `sun_angle(s)`, où `s` est une chaîne de caractère contenant l'heure au format "hh:mm", qui renvoie l'angle en dégrées du soleil au-dessus de l'horizon. Si l'heure d'entrée est avant 6h00 ou après 18h00, la fonction devra retourner "Je ne vois pas le soleil!".

Source : https://py.checkio.org/en/mission/sun-angle/

Correction

```python
def sun_angle(time):
    h,m = time.split(":")
    m_tot = int(h)*60+int(m)
    if m_tot<6*60 or m_tot>18*60:
        return "I don't see the sun!"
    else:
        return 180/((18-6)*60)*(m_tot-6*60)

print(sun_angle("07:00"))
print(sun_angle("12:15"))
print(sun_angle("01:23"))
```

```
15.0
93.75
I don't see the sun!
```

Exercice Bonus 6.122 (Angle entre les aiguilles d'une horloge)

Écrire une fonction qui prend en entrée une chaîne de caractères représentant l'heure au format `HH:MM:SS` et retourne l'angle le plus petit entre les aiguilles des heures et des minutes de l'horloge pour ce moment donné.

L'angle doit être exprimé en degrés, être compris entre 0 et 180 et arrondi à deux décimales près. Par exemple, si l'éntrée est `15:15:30`, la sortie sera `7.50`.

Ne pas oublier de prendre en compte les secondes qui font avancer l'aiguille des heures.

Correction

1. On crée la fonction `angle_aiguilles_horloge` avec comme seul paramètre d'entrée "`heure`" qui sera la chaîne de caractères représentant l'heure au format `HH:MM:SS`.
2. On divise cette chaîne en ses composantes heures, minutes et secondes en utilisant la méthode `split()` et on les convertit en nombres entiers.
3. Si l'heure est ≥ 12, on calcule modulo 12.
4. On procède ensuite au calcul du nombre total de minutes : aux minutes donnés on ajoute une fraction de minute qui est due aux secondes (converties en minutes selon `secondes/60`).
5. Pour calculer l'angle entre les aiguilles des heures et des minutes, on utilise alors la formule suivante :

$$\vartheta = \left| \underbrace{360\left(\frac{\text{heures}}{12} + \frac{\text{total\_minutes}}{60 \times 12}\right)}_{\text{Position de l'aiguille des heures}} - \underbrace{360\left(\frac{\text{total\_minutes}}{60}\right)}_{\text{Pos. aig. minutes}} \right|$$

6. Si l'angle calculé dépasse 180 degrés, on prendra l'angle complémentaire $(360 - \vartheta)$ pour obtenir l'angle le plus petit.

7. Enfin, on retourne l'angle arrondi à deux décimales près.

```
def angle_aiguilles_horloge(heure):
    
    composantes = heure.split(':')
    heures, minutes, secondes = [int(c) for c in composantes]
    
    heures = heures%12
    total_minutes = minutes + secondes / 60

    angle = abs(360 * (heures/12 + total_minutes/60/12) - 360 * (total_minutes/60))
    angle = min( angle , 360 - angle)
    
    return round(angle, 2)

print(angle_aiguilles_horloge("12:00:00"))
print(angle_aiguilles_horloge("12:00:30"))
print(angle_aiguilles_horloge("12:01:00"))
print(angle_aiguilles_horloge("15:15:00"))
```

```
0.0
2.75
5.5
7.5
```

Exercice Bonus 6.123 (Most Wanted Letter)

Dans un texte en anglais, qui contient des lettres et des signes de ponctuation, il faut déterminer la lettre la plus fréquente (*cf.*. exercice 4.48). La lettre renvoyée doit être en casse minuscule. Dans cette recherche, la casse n'a pas d'importance. On considère par exemple, pour compter le nombre de `"a"`, que `"A" == "a"`. Assurez vous de ne compter ni les signes de ponctuation, ni les chiffres, ni les espaces : uniquement les lettres.

Si deux lettres ou plus apparaissent à la même fréquence, il faut renvoyer la liste de ces lettres classées par ordre alphabétique. Par exemple : `"one"` contient `"o"`, `"n"`, `"e"` une fois chacun, donc on doit renvoyer `["e","n","o"]`.

Input : Un texte à analyser comme chaîne de caractères.

Output : La liste des lettres les plus fréquentes en minuscule.

Source : https://py.checkio.org/fr/mission/most-wanted-letter-2/

Correction

```
def myhisto(text: str) -> dict:
    lista = [c for c in text.lower() if c.isalpha()]
    lettre = {}
    for c in lista:
        lettre[c] = lettre.get(c,0)+1
    return lettre

def most_wanted(text: str) -> str:
    lettre = myhisto(text)
    out = [key for key in lettre.keys() if lettre[key] == max(lettre.values())]
    out.sort()
    return out

# TESTS
TEST = [ "Hello World!","How do you do?","One","Oops!","AAaooo!!!!","abe","a" * 9000 + "b" * 1000]

print( *[ f"{most_wanted(text)}" for text in TEST ] , sep = "\n" )
```

```
['l']
['o']
['e', 'n', 'o']
['o']
['a', 'o']
['a', 'b', 'e']
['a']
```

Exercice Bonus 6.124 (Bigger Price)

Vous avez une liste avec tous les produits disponibles dans un magasin. Les données sont représentées sous forme de liste de dictionnaires, chaque dictionnaire représentant un produit avec deux clés: "name" et "price". Le nombre des produits les plus chers sera donné comme premier argument (int) et toutes les données comme second argument. Votre mission ici est de trouver les produits les plus chers (et renvoyer une liste de dictionnaires avec juste ces éléments).

Source: https://py.checkio.org/mission/bigger-price/solve/

Correction

```
bigger_price = lambda n,data : sorted( data , key=lambda x:x["price"] ) [len(data)-n:]

# TESTS
data=[{"name": "bread",      "price": 100},\
      {"name": "wine",       "price": 138},\
      {"name": "meat",       "price": 15},\
      {"name": "water",      "price": 1},\
      {"name": "pen",        "price": 5},
      {"name": "whiteboard", "price": 170}]

# print(bigger_price(2,data))
# print(bigger_price(1,data))

from tabulate import tabulate
print(tabulate(bigger_price(2,data), headers="keys"))
# print(tabulate(bigger_price(1,data), headers="keys"))
```

```
name          price
----------  -------
wine            138
whiteboard      170
```

Exercice Bonus 6.125 (Sum by Types)

Vous avez une liste. Chaque valeur de cette liste peut être une chaîne de caractères ou un entier. Votre tâche ici est de renvoyer deux valeurs. Le premier est une concaténation de toutes les chaînes de la liste donnée. Le second est une somme de tous les entiers de la liste donnée.

Source: https://py.checkio.org/mission/sum-by-type/solve/

Correction

```
def sum_by_types(L):
    s = ''.join([ell for ell in L if type(ell)==str])
    n = sum([ell for ell in L if type(ell)==int])
    return (s, n)

# TESTS
print(sum_by_types([]))
print(sum_by_types([1, 2, 3]))
print(sum_by_types(['1', 2, 3]))
print(sum_by_types(['1', '2', 3]))
print(sum_by_types(['1', '2', '3']))
print(sum_by_types(['size', 12, 'in', 45, 0]))
```

```
('', 0)
('', 6)
('1', 5)
('12', 3)
('123', 0)
('sizein', 57)
```

Exercice Bonus 6.126 (Common Words)

Continuons à examiner les mots. On vous donne deux chaînes de caractères avec des mots séparés par des virgules. Votre fonction doit trouver tous les mots qui apparaissent dans les deux chaînes de caractères. Le résultat doit être représenté comme une chaîne de mots séparés par des virgules dans l'ordre alphabétique.

Préconditions :

- tous les mots sont séparés par des virgules,
- tous les mots sont composés de lettres romaines en minuscule,
- les mots ne sont pas répétés au sein d'une même chaîne de caractères.

Source : https://py.checkio.org/mission/common-words/solve/

Correction

```
def common_words(line1: str, line2: str) -> str:
    L1 = line1.split(',')
    L2 = line2.split(',')
    L3 = [mot for mot in L1 if mot in L2]
    L3.sort()
    if len(L3)!=0:
        return ','.join(L3)
    else:
        return ''

# TESTS
print(common_words('hello,world', 'hello,earth'))
print(common_words('one,two,three', 'four,five,six'))
print(common_words('one,two,three', 'four,five,one,two,six,three'))
```

```
hello

one,three,two
```

Exercice Bonus 6.127 (Flatten list)

Soit en entrée une liste qui contient des entiers ou d'autres listes d'entiers imbriquées. Vous devez mettre toutes les valeurs entières dans une seule liste. L'ordre doit être tel qu'il était dans la liste d'origine.

Exemples :

- `flat_list([1, 2, 3])` renvoie `[1, 2, 3]`
- `flat_list([1, [2, 2, 2], 4])` renvoie `[1, 2, 2, 2, 4]`
- `flat_list([[[2]], [4, [5, 6, [6], 6, 6, 6], 7]])` renvoie `[2, 4, 5, 6, 6, 6, 6, 6, 7]`
- `flat_list([-1, [1, [-2], 1], -1])` renvoie `[-1, 1, -2, 1, -1]`

Source : https://py.checkio.org/en/mission/flatten-list/

Correction

Avec une fonction récursive :

```
def flat_list(array):
    L = []
    for item in array:
        if type(item) == list: # idem que if isinstance(item,list):
            L.extend(flat_list(item))
        else:
            L.append(item)
    return L

# En version one-liner
# f = lambda array : [array] if int==type(array) else sum( map(f,array) , [] )

# TEST
a1 = [1,4,3,2]
a2 = [1,[2,2,2],4]
a3 = [[[2]], [4, [5, 6, [6], 6, 6, 6], 7]]
a4 = [-1,[1,[-2,[3],[[5],[10,-11],[1,100,[-1000,[5000]]],[20,-10,[[[]]]]]]]]
```

```
a5 = [[7007],[6006],[7007]]
a6 = [[[[53],-43],33],23]
a7 = [20,[20,[20,[20]]]]
a8 = [1001,2001,3001,[4001,5001]]
a9 = [10,[20,[30]]]
a10 = [[[22]],[44,[55,66,[66],66,66,66],77]]
a11 = [13,[23,23,23],43]
a12 = [[7],[6],[7]]
a13 = [[[[3],3],3],3]
for array in [a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13]:
    print(f"{array}\n{flat_list(array)}\n")
```

```
[1, 4, 3, 2]
[1, 4, 3, 2]

[1, [2, 2, 2], 4]
[1, 2, 2, 2, 4]

[[[2]], [4, [5, 6, [6], 6, 6, 6], 7]]
[2, 4, 5, 6, 6, 6, 6, 6, 7]

[-1, [1, [-2, [3], [[5], [10, -11], [1, 100, [-1000, [5000]]], [20, -10, [[[]]]]]]]]
[-1, 1, -2, 3, 5, 10, -11, 1, 100, -1000, 5000, 20, -10]

[[7007], [6006], [7007]]
[7007, 6006, 7007]

[[[[53], -43], 33], 23]
[53, -43, 33, 23]

[20, [20, [20, [20]]]]
[20, 20, 20, 20]

[1001, 2001, 3001, [4001, 5001]]
[1001, 2001, 3001, 4001, 5001]

[10, [20, [30]]]
[10, 20, 30]

[[[22]], [44, [55, 66, [66], 66, 66, 66], 77]]
[22, 44, 55, 66, 66, 66, 66, 66, 77]

[13, [23, 23, 23], 43]
[13, 23, 23, 23, 43]

[[7], [6], [7]]
[7, 6, 7]

[[[[3], 3], 3], 3]
[3, 3, 3, 3]
```

Exercice Bonus 6.128 (Plan cyclique)

On considère au plus 26 personnes notées A, B, C, D, E, F ... qui peuvent s'envoyer des messages. Le plan d'envoi doit respecter deux règles :

- chaque personne ne peut envoyer des messages qu'à une seule personne (éventuellement elle-même),
- chaque personne ne peut recevoir des messages qu'en provenance d'une seule personne (éventuellement elle-même).

Il s'agit donc d'une bijection de l'ensemble de personnes dans lui même.

Pour un plan d'envoi donné, s'il existe une suite de personnes dans laquelle la dernière est la même que la première, on dit que c'est un cycle. Lorsqu'un plan d'envoi comporte un unique cycle, on dit que le plan d'envoi est cyclique.

Plan a : Voici un exemple avec 6 personnes de «plan d'envoi des messages»

$$A \to E \to B \to F \to A \qquad C \to D \to C$$

C'est bien un plan car il respecte les deux règles (c'est une bijection). Pour le vérifier, il suffit d'écrire dans une matrice dans la première ligne les personnes qui envoient puis dans la deuxième celle qui reçoivent :

$$\begin{matrix} A & E & B & F & C & D \\ E & B & F & A & D & C \end{matrix}$$

on remarque chaque personne est présente une seule fois dans chaque ligne. De plus, il y a deux cycles distincts : un premier cycle avec A, E, B, F et un second cycle avec C et D.

Le dictionnaire correspondant à ce plan d'envoi est

`plan_a = {'A':'E', 'B':'F', 'C':'D', 'D':'C', 'E':'B', 'F':'A'}`

Plan b : Le plan d'envoi suivant

`plan_b = {'A':'C', 'B':'F', 'C':'E', 'D':'A', 'E':'B', 'F':'D'}`

comporte un unique cycle : $A \to C \to E \to B \to F \to D \to A$. Dans ce cas le plan d'envoi est cyclique.

Pour savoir si un plan d'envoi de messages comportant N personnes est cyclique, on peut utiliser l'algorithme ci-dessous :

- on part d'un expéditeur et on inspecte son destinataire dans le plan d'envoi,
- chaque destinataire devient à son tour expéditeur, selon le plan d'envoi, tant qu'on ne « retombe » pas sur l'expéditeur initial,
- le plan d'envoi est cyclique si on l'a parcouru en entier.

Correction

Sujet 38.2 : https://glassus.github.io/terminale_nsi/T6_6_Epreuve_pratique/BNS_2023/?s=03

```python
def est_cyclique(plan):
    '''
    Prend en paramètre un dictionnaire `plan` correspondant à un plan d'envoi de messages
    (ici entre les personnes A, B, C, D, E, F).
    Renvoie True si le plan d'envoi de messages est cyclique et False sinon.
    '''

    # si le plan ne respecte pas les règles
    if sorted(plan.keys()) != sorted(plan.values()):
        return False

    expediteur = 'A'
    destinataire = plan[expediteur]
    nb_destinaires = 1

    while destinataire != expediteur:
        destinataire = plan[destinataire]
        nb_destinaires += 1

    return nb_destinaires == len(plan)

#tests
print(est_cyclique({'A':'B', 'B':'C'})) # no plan
print(est_cyclique({'A':'B', 'B':'A'})) # plan cyclique
print(est_cyclique({'A':'B', 'B':'A', 'C':'C'})) # plan 2 cycles
print(est_cyclique({'A':'E', 'F':'A', 'C':'D', 'E':'B', 'B':'F', 'D':'C'})) # plan 2 cycles
print(est_cyclique({'A':'E', 'F':'C', 'C':'D', 'E':'B', 'B':'F', 'D':'A'})) # plan cyclique
print(est_cyclique({'A':'B', 'F':'C', 'C':'D', 'E':'A', 'B':'F', 'D':'E'})) # plan cyclique
print(est_cyclique({'A':'B', 'F':'A', 'C':'D', 'E':'C', 'B':'F', 'D':'E'})) # plan 2 cycles
```

```
False
True
False
False
True
```

```
True
False
```

Exercice Bonus 6.129 (Systèmes Électoraux)

Les systèmes électoraux sont des méthodes de vote utilisées pour élire des représentants ou prendre des décisions collectives. Trois systèmes électoraux courants sont souvent étudiés :

1. **Système de vote majoritaire à un tour (Plurality)** : le candidat avec le plus de votes remporte l'élection.
2. **Système de vote par approbation** : chaque électeur peut voter pour autant de candidats qu'il le souhaite, et le candidat avec le plus de votes l'emporte.
3. **Système de vote par rangement (Vote préférentiel)** : les électeurs classent les candidats par ordre de préférence. Le gagnant est déterminé selon un algorithme qui prend en compte ces classements.

Implémentez chaque système électoral et testez vos implémentations avec des exemples de votes et comparez les résultats obtenus.

Correction

```python
def vote_plurality(votes):
    vote_count = {}
    for vote in votes:
        if vote in vote_count:
            vote_count[vote] += 1
        else:
            vote_count[vote] = 1
    winner = max(vote_count, key=vote_count.get)
    return winner

def vote_approval(votes):
    vote_count = {}
    for vote in votes:
        for candidate in vote:
            if candidate in vote_count:
                vote_count[candidate] += 1
            else:
                vote_count[candidate] = 1
    max_approval = max(vote_count.values())
    winners = [candidate for candidate, count in vote_count.items() if count == max_approval]
    return winners

def vote_ranking(votes):
    rankings = {}
    for vote in votes:
        for i, candidate in enumerate(vote):
            if candidate in rankings:
                rankings[candidate] += i
            else:
                rankings[candidate] = i
    winner = min(rankings, key=rankings.get)
    return winner
```

```python
# Test comparatif
votes = ['A', 'B', 'A', 'C', 'B', 'A', 'A', 'C', 'B', 'B']

winner_plurality = vote_plurality(votes)
print("Winner (Plurality):", winner_plurality)


votes = [['A','C'], ['B'], ['A','B','C'], ['A','C'], ['B'], ['A'], ['A'], ['C'], ['B'], ['B']]
winners_approval = vote_approval(votes)
print("Winners (Approval):", winners_approval)
```

```
ranked_votes = [['A', 'B', 'C'], ['B', 'C', 'A'], ['C', 'B', 'A'], ['A', 'C', 'B']]
winner_ranking = vote_ranking(ranked_votes)
print("Winner (Ranking):", winner_ranking)
```

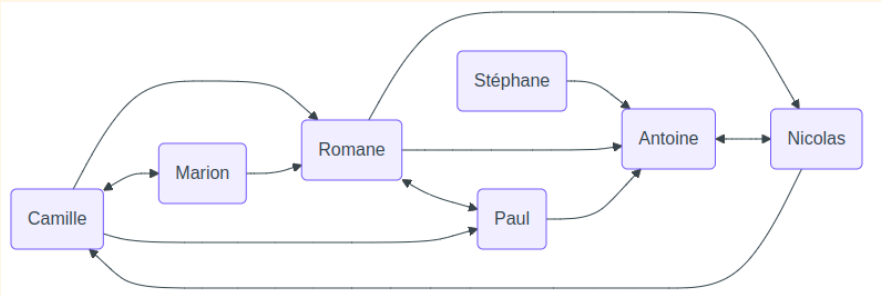
```
Winner (Plurality): A
Winners (Approval): ['A', 'B']
Winner (Ranking): A
```

Exercice Bonus 6.130 (CodEx: Circulation de rumeurs)

On modélise la circulation de rumeurs dans un groupe par un graphe orienté, où chaque arête indique une relation de transmission de rumeurs entre deux personnes. Par exemple, "Romane → Antoine" signifie que Romane informe Antoine, mais pas l'inverse, tandis que "Antoine ↔ Nicolas" indique une relation bidirectionnelle.

Ce graphe est représenté par un dictionnaire Python où chaque clé est un nom et chaque valeur est la liste des personnes informées par cette clé.

```
graphe = {
    'Camille':  ['Romane', 'Marion', 'Paul'],
    'Romane':   ['Nicolas', 'Antoine', 'Paul'],
    'Marion':   ['Camille', 'Romane'],
    'Paul':     ['Antoine', 'Romane'],
    'Antoine':  ['Nicolas'],
    'Nicolas':  ['Camille', 'Antoine'],
    'Stéphane': ['Antoine']
}
```



Connaissant la structure du graphe et l'origine de la rumeur (la première personne informée), on cherche à savoir combien de fois celle-ci a été racontée avant d'atteindre une personne donnée. Par exemple, si la rumeur part de Romane, elle sera racontée 1 seule fois avant qu'Antoine ne l'apprenne, 3 fois avant que Marion ne l'apprenne.

Écrire une fonction `distance` qui, donnée une origine (le nom de la personne à l'origine de la rumeur), une destination (le nom d'une personne) et le graphe (un dictionnaire), retourne le nombre minimal de transmissions pour atteindre la destination. Si la destination est inaccessible, la fonction renvoie `None`.

Source : https://codex.forge.apps.education.fr/exercices/rumeur/

Correction

La fonction `distance` utilise une recherche en largeur (BFS pour *Breadth-First Search*) pour déterminer le nombre minimal de transmissions. On initialise une file avec la personne d'origine et une distance de 0, puis on explore les relations jusqu'à atteindre la destination ou épuiser les options.

1. *Initialisation* :
   - On part de la personne d'origine, avec une file (queue) contenant cette personne et une distance de 0.
   - Un ensemble garde trace des personnes déjà visitées.
2. *Exploration* :
   - Tant que la file n'est pas vide, retirer la première personne.
   - Si cette personne est la destination, retourner la distance et le chemin.
   - Sinon, la marquer comme visitée et ajouter ses relations à la file avec une distance +1, en construisant le chemin.
3. *Terminaison* : Si la file est vide sans atteindre la destination, retourner `None`.

Prenons l'exemple où la rumeur part de Romane pour atteindre Antoine.

- *Initialisation*:
  - Origine : Romane (index 1)
  - Destination : Antoine (index 4)
  - File : [(1, 0, ['Romane'])] (Romane, distance 0, chemin avec l'origine)
  - Visité : {}
- *Exploration*:
  - Retirer Romane de la file : (1, 0, ['Romane'])
  - Marquer Romane comme visitée : Visité = {1}
  - Ajouter les relations de Romane (Nicolas, Antoine, Paul) : File = [(5, 1, ['Romane', 'Nicolas']), (4, 1, ['Romane', 'Antoine']), (3, 1, ['Romane', 'Paul'])]
- *Exploration (suite)*:
  - Retirer Nicolas : (5, 1, ['Romane', 'Nicolas'])
  - Marquer Nicolas comme visité : Visité = {1, 5}
  - Ajouter les relations de Nicolas (Camille, Antoine) : File = [(4, 1, ['Romane', 'Antoine']), (3, 1, ['Romane', 'Paul']), (0, 2, ['Romane', 'Nicolas', 'Camille']), (4, 2, ['Romane', 'Nicolas', 'Antoine'])]
  - Retirer Antoine : (4, 1, ['Romane', 'Antoine'])
  - Antoine est la destination, retourner : (1, ['Romane', 'Antoine'])

```python
def distance(graphe, origine, destination):
    queue = [(origine, 0, [origine])]
    visited = set()

    while queue:
        current_person, current_distance, path = queue.pop(0)

        if current_person == destination:
            return current_distance, path

        visited.add(current_person)

        for relation in graphe.get(current_person, []):
            if relation not in visited:
                queue.append((relation, current_distance + 1, path + [relation]))

    return None

# Exemple d'utilisation
graphe = {
    'Camille':  ['Romane', 'Marion', 'Paul'],
    'Romane':   ['Nicolas', 'Antoine', 'Paul'],
    'Marion':   ['Camille', 'Romane'],
    'Paul':     ['Antoine', 'Romane'],
    'Antoine':  ['Nicolas'],
    'Nicolas':  ['Camille', 'Antoine'],
    'Stéphane': ['Antoine']
}

print(distance(graphe, 'Romane', 'Antoine'))  # Output: (1, ['Romane', 'Antoine'])
print(distance(graphe, 'Romane', 'Marion'))   # Output: (3, ['Romane', 'Nicolas', 'Camille', 'Marion'])
print(distance(graphe, 'Stéphane', 'Paul'))   # Output: None
```

```
(1, ['Romane', 'Antoine'])
(3, ['Romane', 'Nicolas', 'Camille', 'Marion'])
(4, ['Stéphane', 'Antoine', 'Nicolas', 'Camille', 'Paul'])
```

CHAPITRE 7

Modules

Un **module** est une **collection de fonctions prédéfinies qui peuvent être importées** dans un programme pour être utilisées selon les besoins. Ce mécanisme offre une solution pratique pour éviter la réécriture de code commun et pour bénéficier d'une panoplie de fonctionnalités avancées déjà implémentées. En utilisant des modules, les développeurs peuvent importer des fonctionnalités existantes et ainsi étendre facilement les capacités de leur programme. En parallèle des modules, les fichiers Python peuvent être structurés en un agencement de dossiers formant un "package". Un **package** consiste en un module contenant d'autres modules, créant ainsi une structure hiérarchique grâce à des sous-packages.

Il y a différents types de modules et de packages : certains sont inclus dans la distribution standard de Python, comme `random` ou `math` ; d'autres peuvent être ajoutés, tels que `numpy` ou `matplotlib` ; et il est également possible d'en créer soi-même, généralement sous la forme d'un fichier Python contenant un ensemble de fonctions. Python dispose d'une bibliothèque très riche de modules, englobant des centaines de modules prêts à l'emploi dans divers domaines tels que les mathématiques, l'administration système, la programmation réseau, la manipulation de fichiers, etc. L'utilisation de ces modules évite de réinventer la roue à chaque programme, accélérant considérablement le processus de développement et de maintenance du code. Pour explorer les modules et packages disponibles, vous pouvez consulter le site https://pypi.org/ (The Python Package Index).

7.1. Importation d'un module

Pour utiliser les fonctions d'un module, l'importation est la première étape nécessaire. Nous allons voir trois méthodes pour importer puis utiliser ces fonctions (et une quatrième, la plus part du temps déconseillée).

Méthodes 1 et 2.

Méthode 1.
```
# importation
import ModuleName
# Les fonctions sont utilisées en préfixant leur nom avec le nom du module
ModuleName.FunctionName(parameters)
```

Méthode 2.
```
# importation avec un alias pour le Nom du Module
import ModuleName as alias
# Les fonctions  sont utilisées en préfixant leur nom avec l'alias du module
alias.FunctionName(parameters)
```

Une fois importé, il est possible :

- d'afficher la liste des fonctions définies dans le module avec `dir(ModuleName)`,
- d'obtenir de l'aide sur le module et la description de chaque fonction avec `help(ModuleName)`,
- d'obtenir la description d'une seule fonction avec `help(FunctionName)`.

Remarque 14 (Pourquoi indiquer le nom du module lors des appelles d'une fonction de ce module ?)
Un problème récurrent en programmation est celui de la collision de noms. Écrire un programme implique de nommer les différents éléments du programme (variables, paramètres, fonctions, objets...). Si on désire réutiliser des fonctions développées par d'autres personnes, il est possible que l'on soit amené à incorporer des fonctions qui portent le même nom. En Python, si on déclare une fonction avec un nom déjà utilisé pour désigner une autre fonction, cette dernière ne sera plus accessible. On se retrouve confronté à un phénomène de collision des noms. Pour résoudre cette situation, la solution la plus couramment utilisée consiste à créer des espaces de noms différents pour les deux fonctions. En Python, un module porte un nom qui identifie un espace dans lequel on peut définir des fonctions. Ainsi si les modules module1 et le module2 définissent tous les deux une fonction s'appelant f(), il sera possible de les distinguer à l'appel en préfixant le nom de la fonction par le nom du module.

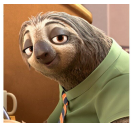
Méthode 3. Importation sélective de quelques fonctions

Parfois on ne souhaite pas importer tout un module et il peut être fastidieux de préfixer systématiquement un nom par le module qui le définit. Grâce à la syntaxe suivante, les fonctions peuvent être utilisées directement :

```
# importation de quelques fonctions spécifiques
from ModuleName import function1, function2
# Les fonctions sont utilisées sans prefix
function1(parameters)
```

Méthode 4 (dangéreuse). Importation avec le "Mode Paresseux"

Il existe une quatrième méthode pour importer un module et ses fonctions, que j'ai appelée le "Mode Paresseux" :



```
# importation de toutes les fonctions du module
from ModuleName import *
# Les fonctions sont utilisées sans prefix
FunctionName(parameters)
```

Cette méthode d'importation est déconseillée, car en important l'intégralité du contenu d'un module sans prefix, elle peut mener à des conflits si deux modules définissent une fonction identique. Ce n'est pas si rare d'avoir des fonctions qui portent le même nom dans des modules différents. Des exemples courants abondent, comme la fonction `plot()` définie dans des modules tels que `Matplotlib`, `SymPy`, `Seaborn` et bien d'autres. Un autre cas concret concerne le module `scipy` qui définit deux fonctions informatiques `gamma`: l'une dans `scipy.special` pour la fonction mathématique gamma, l'autre dans `scipy.stats` pour la distribution gamma:

```
scipy.special.gamma # gamma function
scipy.stats.gamma # gamma distribution
```

Bien qu'il puisse sembler tentant de simplifier l'importation des fonctions et d'économiser du temps dans leur invocation (surtout si vous prévoyez initialement d'utiliser un seul module dans votre script), l'utilisation de cette méthode peut conduire à des conflits si vous commencez à utiliser plusieurs modules. **Il est donc recommandé de suivre dès le début les bonnes pratiques et d'éviter cette stratégie dictée par la facilité.** Préférez plutôt l'importation explicite des fonctions pour éviter ces problèmes potentiels.

EXEMPLE (EXEMPLE DE CONFLIT AVEC LE MODE PARESSEUX)

Prenons l'exemple des fonctions mathématiques regroupées dans les modules `math` et `cmath`, ce dernier étant spécialisé pour les nombres complexes. Les deux modules contiennent une fonction `sqrt` pour la racine carrée. Avec le mode paresseux, Python utilisera **celle du dernier module importé**.

```
>>> from cmath import *
>>> from math import *
>>> print('Racine carrée de -4 =', sqrt(-4))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

```
>>> from math import *
>>> from cmath import *
>>> print('Racine carrée de -4 =', sqrt(-4))
Racine carrée de -4 = 2j
```

Si on utilise une des 3 méthodes préconisées, pas de risque de conflit car l'appel à la fonction `sqrt` est explicite et provient du module `cmath`, évitant ainsi les erreurs d'exécution.

```
>>> import cmath
>>> import math
>>> print('Racine carrée de -4 =', cmath.sqrt(-4))
Racine carrée de -4 = 2j
```

```
>>> from cmath import sqrt
>>> import math
>>> print('Racine carrée de -4 =', sqrt(-4))
Racine carrée de -4 = 2j
```

Résumé

L'instruction `import` peut être placée à n'importe quel endroit dans un programme (y compris dans le corps d'une fonction). Cela permet d'importer le module uniquement lorsqu'il est réellement nécessaire d'accéder à un élément qu'il définit. Mais je vous conseil de la placer toujours au début du fichier.

Si vous souhaitez importer plusieurs modules, vous pouvez même utiliser une seule instruction `import` en séparant le nom des modules par une virgule, comme par exemple `import os, sys, random`. cependant, on importe en général un module par ligne. D'abord les modules internes (classés par ordre alphabétique), c'est-à-dire les modules de base de Python, puis les modules externes (ceux que vous avez installés en plus). Si le nom du module est trop long, on peut utiliser un alias. L'instruction `from` est tolérée si vous n'importez que quelques fonctions clairement identifiées.

```
import module_interne_1, module_interne_2
import module_interne_3_qui_a_un_nom_long as mod3
from module_interne_4 import fonction_spécifique
```

```python
from module_interne_5 import constante_1, fonction_1, fonction_2

import module_externe_1
import module_externe_2
```

7.2. Quelques modules mathématiques courants

Il existe une série de modules que vous serez probablement amenés à utiliser si vous programmez en Python. En particulier, parmi les modules/packages essentiels pour des étudiants en licence de mathématiques on trouve

math	numpy	pandas	scipy	sympy	matplotlib
		pandas		SymPy	

Ce cours d'introduction à la programmation pour les Mathématiques en Python abordera certains d'entre eux, tandis que les autres seront explorés au fil des prochains semestres de la Licence Mathématiques à l'Université de Toulon.

7.2.1. Le module `math` : fonctions et constantes mathématiques de base

En Python, seules quelques fonctions mathématiques sont pré-définies :

`abs(a)`	Valeur absolue de a
`max(suite)`	Plus grande valeur de la suite
`min(suite)`	Plus petite valeur de la suite
`round(a,n)`	Arrondi de a à n décimales près
`pow(a,n)`	Exponentiation, renvoie a^n, équivalent à `a**n`
`sum(L)`	Somme des éléments de la suite
`divmod(a,b)`	Renvoie le quotient et le reste de la division de a par b
`cmp(a,b)`	Renvoie $\begin{cases} -1 & \text{si } a < b, \\ 0 & \text{si } a = b, \\ 1 & \text{si } a > b. \end{cases}$

Remarque à propos de `round()` : sans le deuxième argument, elle arrondit a à l'entier le plus proche. Il est important de noter que pour les nombres exactement à mi-chemin entre deux entiers (par exemple, 0.5), elle arrondit au nombre pair le plus proche. Par exemple, `round(2.5)` serait égal à 2, tandis que `round(3.5)` serait égal à 4. Cela est dû à une convention d'arrondi appelée "arrondi par la parité".

D'autres fonctions mathématiques sont définies dans le module `math`. Comme mentionné précédemment, plusieurs syntaxes sont disponibles pour importer des fonctions à partir d'un module :

Méthode 1

```python
import math
print(math.sin(math.pi)) # sin(π)
```

1.2246467991473532e-16

Méthode 2

```python
import math as mm
print(mm.sin(mm.pi))
```

1.2246467991473532e-16

Méthode 3

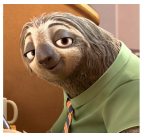
```python
from math import sin, pi
print(sin(pi))
```

1.2246467991473532e-16

Mode "paresseux"

```python
from math import *
print(sin(pi))
```

1.2246467991473532e-16



Voici la liste des fonctions définies dans le module `math` (pour une description complète utiliser `print(help(math))`) :

```python
import math
print(dir(math))
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh',
↪ 'atan', 'atan2', 'atanh', 'cbrt', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e',
↪ 'erf', 'erfc', 'exp', 'exp2', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
↪ 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp',
↪ 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod',
↪ 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'sumprod', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

Notons que le module définit les deux constantes π et e.

```
import math
print(math.pi, math.e, math.exp(1))
```

```
3.141592653589793 2.718281828459045 2.718281828459045
```

Attention, les fonctions `prod` et `dist` ne sont disponibles que depuis la version 3.8 de Python.

```
import math
print(math.prod([1,2,3,4]))

p = [3, 3]
q = [6, 12]
# Calculate Euclidean distance
print(math.dist(p, q)) # = √((x_p − x_q)² + (y_p − y_q)²)
```

```
24
9.486832980505138
```

Remarque 15
L'arithmétique à virgule flottante est sujette à un débordement (*overflow*) si une valeur devient trop grande. Cela provoque un type d'erreur (*exception*) qui, si elle n'est pas traitée, interrompt l'exécution du programme :

```
>>> import math
>>> math.exp(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: math range error
```

En revanche, la taille des nombres entiers n'est limitée que par la quantité de mémoire disponible, ainsi 1000! peut être évalué avec précision à l'aide de la fonction `math.factorial` :

```
>>> import math
>>> math.factorial(1000)
402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404800479988610197
```

7.2.2. Le module numpy (algèbre matricielle)

Le module `numpy` est la boîte à outils indispensable pour faire du calcul scientifique avec Python. Il permet d'effectuer des calculs sur des vecteurs (ou des matrices ou plus généralement les tableaux à n dimensions), élément par élément, via un nouveau type d'objet appelé `array`.

Traditionnellement, on charge la totalité du module `numpy` de la manière suivante :

```
import numpy as np
```

On évitera de charger `numpy` par "`from numpy import *`" : le nombre de fonctions importées est en effet trop important et avec lui le risque d'homonymie avec les définitions déjà présentes au moment de l'importation.

La différence majeure avec les listes, que nous avons utilisé jusqu'ici pour représenter des vecteurs (ou des matrices) est que les tableaux `numpy` sont homogènes, c'est-à-dire constitués d'éléments du même type.

La fonction `array()` convertit un container (comme une liste ou un tuple) en un objet de type `array`. Voici un exemple simple de conversion d'une liste à une dimension en objet `array` :

```
>>> import numpy as np
>>> L = [1, 2, 3]
>>> A = np.array(L)
>>> print(f"{L = }, {type(L) = },\n{A = }, {type(A) = }")
L = [1, 2, 3], type(L) = <class 'list'>,
A = array([1, 2, 3]), type(A) = <class 'numpy.ndarray'>
```

La différence fondamentale entre un objet array à une dimension et une liste (ou un tuple) est que celui-ci est considéré comme un vecteur. Par conséquent, on peut effectuer des opérations élément par élément sur ce type d'objet, ce qui est bien commode lorsqu'on analyse de grandes quantités de données. Regardez ces exemples :

```
>>> f = lambda x : x**2
>>> # L2 = f(L) # ERROR !
>>> L2 = [ f(x) for x in L ]
>>> A2 = f(A)
>>> print(f"{L2 = }, {A2 = }")
L2 = [1, 4, 9], A2 = array([1, 4, 9])
```

Remarque 16 (Fonction vectorisée)
Une fonction vectorisée (“universelle” ou «ufunc», abréviation de *universal function*, est le terme exacte) est une fonction qui peut s'appliquer terme à terme aux éléments d'un tableau. Si f est une ufunc et si $a = [a_0, a_1, \ldots, a_{n-1}]$ est un tableau, alors $f(a)$ renvoie le tableau $[f(a_0), f(a_1), \ldots, f(a_{n-1})]$. Un grand nombre de fonctions usuelles sont directement «universalisées» dans `numpy`. Les fonctions mathématiques gardent le même nom (préfixé par `np` si on a importé `numpy` par `import numpy as np`).

```
import numpy as np
xx = np.arange(4) # comme range mais produit un array
yy = np.sin(xx*np.pi/2)
print(f"{xx = },\n{yy = }")
```

```
xx = array([0, 1, 2, 3]),
yy = array([ 0.0000000e+00,  1.0000000e+00,  1.2246468e-16, -1.0000000e+00])
```

EXEMPLE
Ci-dessous on compare les performances de deux approches différentes pour calculer une fonction mathématique sur un grand ensemble de valeurs. En général, les opérations vectorisées de `numpy` sont beaucoup plus rapides que les boucles Python pures pour des calculs sur des tableaux de données en raison de leur implémentation optimisée en C.

```
import time
import math
import numpy as np

N = 1_000_000

def boucle_math():
    f = lambda t : 2*math.pi**0.25/math.sqrt(3) * (1 - t**2) * math.exp(-(t**2) / 2)
    xs = np.linspace(-5, 5, N)
    return xs, [f(x) for x in xs]

def boucle_np():
    f = lambda t : 2*np.pi**0.25/np.sqrt(3) * (1 - t**2) * np.exp(-(t**2) / 2)
    xs = np.linspace(-5, 5, N)
    return xs, f(xs)

def mesurer_temps(f):
    """Appelle f, mesure et renvoie le nombre de secondes que prend l'exécution de f()"""
    temps_debut = time.time()
    f()
    temps_fin = time.time()
    return temps_fin - temps_debut

temps_f_maths = mesurer_temps(boucle_math)
temps_f_np = mesurer_temps(boucle_np)
print(f"La version Numpy est {temps_f_maths / temps_f_np:.1f} fois plus rapide")
```

```
La version Numpy est 24.5 fois plus rapide
```

Vous étudierez ce module (et certains de ses sous-modules) en L2. Dans ce cours nous nous limiterons à l'utiliser pour ses fonctions vectorisées en combinaison avec le module `matplotlib` (*cf.* chapitre 8).
https://perso.univ-perp.fr/langlois/images/pdf/mp/www.mathprepa.fr-une-petite-reference-numpy.pdf
https://python.sdv.univ-paris-diderot.fr/17_modules_interet_bioinfo/

Remarque 17 (Fonctions trigonométriques dans différents modules)
Les langages de programmation utilisent des noms différents pour les fonctions qu'ils prennent en charge. Par exemple, les fonctions arcsin, arccos et arctan s'appellent `asin acos atan` dans le module `math` et `arcsin arccos arctan` dans le module `numpy`. Depuis 2009, la norme ISO 80000-2 indique qu'il faut utiliser le prefix `arc`.

7.2.3. Le module scipy (calcul approché)

Selon la convention, scipy est habituellement importé en utilisant l'alias sp.

Cette librairie contient de nombreux algorithmes très utilisés par les personnes qui font du calcul scientifique : fft, algèbre linéaire (méthodes directes ou itératives pour résoudre des systèmes linéaires, ...), interpolation, intégration numérique, statistiques et autres algorithmes numériques. On peut voir ce module comme une extension de Numpy car **il contient toutes les fonctions de** Numpy.

Le site de la documentation en fournit la liste : http://docs.scipy.org/doc/scipy/reference

Voici deux exemples d'utilisation : le calcul approché d'une solution d'une équation et le calcul approché d'une intégrale.

fsolve Si on ne peut pas calculer analytiquement la solution d'une équation, on peut l'approcher numériquement. Tout d'abord on définit une fonction f telle que $f(x) = 0$ ssi x est solution de l'équation donnée, on se donne ensuite un point x_0 pas trop éloigné de la solution cherchée et on utilise la fonction fsolve du module scipy.optimize dont la syntaxe est fsolve(f,x0). Cette fonction renvoie une liste contenant la solution (approchée) et une estimation de l'erreur.

Voici un exemple : on cherche à calculer la solution de l'équation $x = \cos(x)$. Une étude des fonction $x \mapsto x$ et $x \mapsto \cos(x)$ montre que la solution se trouve entre 0 et $\pi/2$. On peut donc poser $f(x) = x - \cos(x)$ et $x_0 = 1$:

```
from math import cos
from scipy.optimize import fsolve

x0 = 1
sol = fsolve( lambda x: x-cos(x) , x0 )[0] # On ne garde que la solution approchée
print(sol)
```

```
0.7390851332151607
```

integrate.quad Pour approcher la valeur numérique d'une intégrale on peut utiliser la fonction quad du sous-module integrate du module scipy https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html

```
from scipy import integrate # on importe toutes les fonctions du sous-module integrate

a = 0
b = 1
f = lambda x : x**2
integr = integrate.quad(f,a,b) # On approche ∫_a^b f(x) dx avec f(x) = x^2
print("Integrale =",integr[0], " Erreur =",integr[1] )
```

```
Integrale = 0.33333333333333337  Erreur = 3.700743415417189e-15
```

```
from scipy import exp, sqrt, pi, inf
from scipy.integrate import quad

f = lambda x : exp(-x*x)
app = quad(f, 0,inf)[0] # On approche ∫_0^∞ e^{-x^2} dx
print(app)
print(sqrt(pi)/2) # Valeur exacte de cette intégrale
```

```
0.8862269254527579
0.8862269254527579
```

7.2.4. Le module mpmath

Le module mpmath est une bibliothèque de calcul en haute précision pour les nombres flottants. Il permet d'effectuer des calculs avec une précision arbitraire, ce qui est particulièrement utile pour les applications scientifiques et les calculs nécessitant une grande exactitude.

Avec mpmath, vous pouvez définir des nombres avec une précision beaucoup plus élevée que celle offerte par les types de données flottants natifs de Python. Par exemple :

```
from mpmath import mp

# Définir la précision à 50 chiffres
mp.dps = 50
```

```
# Calculer π avec la précision définie
pi_high_precision = mp.pi
print(pi_high_precision)
```

```
3.1415926535897932384626433832795028841971693993751
```

Ce module fournit une précision arbitraire, mais les calculs restent approchés, simplement ils sont effectués avec plus de chiffres significatifs. Pour obtenir un résultat précis on devra utiliser une bibliothèque qui supporte le calcul abstrait.

7.2.5. Le module `fractions`

Mis à part les limitations dues à la mémoire, on peut considérer que les calculs dans $\mathbb{N}$ sont exacts avec Python. En revanche, les calculs dans $\mathbb{R}$ sont toujours approchés car on travaille avec des nombres flottants, ce qui peut entraîner une perte de précision. Pour travailler dans $\mathbb{Q}$, avec des fractions donc, il est possible d'utiliser un module dédié qui permet de réaliser des calculs exacts. Il s'agit du module `Fraction`.

Le module `Fraction` permet de représenter des fractions et offre la possibilité d'effectuer des opérations sur celles-ci.

```
from fractions import Fraction
print(Fraction(numerator=1, denominator=3), Fraction(2, 10))
```

```
1/3 1/5
```

Les opérations mathématiques de base peuvent être effectuées sur des instances de `Fraction` de la même manière que sur d'autres types de données. Par exemple :

```
from fractions import Fraction
print(1/6 + 2/5, Fraction(1, 6) + Fraction(2, 5))
print(1/6 * 2/5, Fraction(1, 6) * Fraction(2, 5))
```

```
0.5666666666666667 17/30
0.06666666666666667 1/15
```

Dans certains cas on peut obtenir des calculs exactes même dans $\mathbb{R}$ en utilisant le module `sympy` dédié au calcul symbolique. Contrairement aux calculs numériques, qui traitent des valeurs numériques spécifiques, les calculs symboliques permettent de manipuler des expressions mathématiques de manière abstraite. Cela inclut la simplification d'expressions, la résolution d'équations algébriques, le calcul de dérivées et d'intégrales, ainsi que bien d'autres opérations.

7.2.6. ★ Le module `SymPy` (calcul formel)

`SymPy` est une bibliothèque Python pour les mathématiques symboliques. Elle prévoit devenir un système complet de calcul formel ("CAS" en anglais : *Computer Algebra System*) tout en gardant le code aussi simple que possible afin qu'il soit compréhensible et facilement extensible.

- Quelques commandes : https://www.sympygamma.com/
- Pour tester en ligne : https://live.sympy.org/

Avec `sympy`, vous pouvez définir des symboles et effectuer des opérations sur ceux-ci comme vous le feriez avec des variables en algèbre. Par exemple :

```
import sympy as sym
sym.var('x,y')
```

◇

```
# Calculs
s = (x+y)+(x-y)
print(s)
```

$$2x$$

◇

```
# Simplifications
expr = sym.sin(x)**2 + sym.cos(x)**2
print(expr,"=",sym.simplify(expr))
```

$$\sin^2(x) + \cos^2(x) = 1$$

```
expr = (x**3 + x**2 - x - 1)/(x**2 + 2*x + 1)
print(expr,"=",sym.simplify(expr))
```

$$\frac{x^3 + x^2 - x - 1}{x^2 + 2x + 1} = x - 1$$

◇

```
# Fractions
expr = (x**3-y**3)/(x**2-y**2)
print(expr,"=",sym.cancel(expr))
```

$$\frac{x^3 - y^3}{x^2 - y^2} = \frac{x^2 + xy + y^2}{x + y}$$

◇

Encore un exemple :

```
import sympy as sym
sym.var('x')
functions = [sym.sin(x), sym.cos(x), sym.tan(x)]
for f in functions:
    d = sym.Derivative(f, x)
    i = sym.Integral(f, x)
    print(d, "=" , d.doit(), "\t" , i , "=" , i.doit())
```

$$\frac{d}{dx}\sin(x) = \cos(x) \qquad \int \sin(x)\,dx = -\cos(x)$$
$$\frac{d}{dx}\cos(x) = -\sin(x) \qquad \int \cos(x)\,dx = \sin(x)$$
$$\frac{d}{dx}\tan(x) = \tan^2(x) + 1 \qquad \int \tan(x)\,dx = -\log(\cos(x))$$

Si vous utilisez spyder, vous pouvez remplacer `print(...)` par `display(...)`.

◇

Nous allons comparer les modules `scipy`, `mpmath` et `sympy` pour le calcul de π en utilisant la formule

$$\left(\int_{-\infty}^{+\infty} e^{-x^2}\,dx\right)^2.$$

Ces exemples montrent les différentes approches pour le calcul de π en utilisant une intégrale définie. Chacune de ces méthodes a ses propres avantages en termes de précision et de performance.

- Le module `scipy` utilise des méthodes numériques pour évaluer l'intégrale. La précision dépend des algorithmes numériques utilisés (ici celle par default).
```
import scipy.integrate as integrate
import numpy as np
f = lambda x : np.exp(-x**2)
result, error = integrate.quad(f, -np.inf, np.inf)
pi_estimate = result**2
print(pi_estimate)
3.1415926535897927
```
- Le module `mpmath` permet d'effectuer des calculs avec une précision arbitraire et contient une fonction analogue pour l'intégration numérique :
```
from mpmath import mp, quad
mp.dps = 50 # Définir la précision à 50 chiffres
f = lambda x : mp.exp(-x**2)
result = quad(f, [-mp.inf, mp.inf])
pi_estimate = result**2
print(pi_estimate)
3.1415926535897932384626433832795028841971693993751
```
- Le module `sympy` permet de réaliser des calculs symboliques, fournissant une représentation exacte de π sans approximation numérique :
```
import sympy as sp
x = sp.symbols('x')
f = sp.exp(-x**2)
integral = sp.integrate(f, (x, -sp.oo, sp.oo))
pi_estimate = integral**2
print(pi_estimate)
pi
```

7.3. Quelques autres modules courants

7.3.1. Le module `random` : génération de nombres aléatoires

Ce module propose diverses fonctions permettant de générer des nombres (pseudo-)aléatoires qui suivent différentes distributions mathématiques. Il apparaît assez difficile d'écrire un algorithme qui soit réellement non-déterministe (c'est-à-dire qui produise un résultat totalement imprévisible). Il existe cependant des techniques mathématiques permettant de simuler plus ou moins bien l'effet du hasard.

Voici la liste des fonctions fournies par ce module. Noter que, sauf indication contraire, tous les tirages sont réalisés selon une loi uniforme (si on tire beaucoup de nombres, on aura la même probabilité d'obtenir tous les nombres possibles entre les bornes indiquées). Voir la documentation du module pour d'autres lois. [1]

```
import random
print(dir(random))
```

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom', 'TWOPI',
↪  '_ONE', '_Sequence', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
↪  '__name__', '__package__', '__spec__', '_accumulate', '_acos', '_bisect', '_ceil', '_cos', '_e',
↪  '_exp', '_fabs', '_floor', '_index', '_inst', '_isfinite', '_lgamma', '_log', '_log2', '_os', '_pi',
↪  '_random', '_repeat', '_sha512', '_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn',
↪  'betavariate', 'binomialvariate', 'choice', 'choices', 'expovariate', 'gammavariate', 'gauss',
↪  'getrandbits', 'getstate', 'lognormvariate', 'normalvariate', 'paretovariate', 'randbytes',
↪  'randint', 'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform',
↪  'vonmisesvariate', 'weibullvariate']
```

Nous utiliserons en particulier les fonctions suivantes :

`randrange(p,n)`	choisit un entier aléatoirement dans la liste `range(p,n)`, *i.e.* dans la liste `[p,p+1,...,p+n-1]`
`randrange(p,n,h)`	choisit un éléments aléatoirement dans la liste `range(p,n,h)`
`randint(a,b)`	choisit un *entier* aléatoirement dans l'intervalle $[a;b]$ (alias pour `random.randrange(a,b+1)`)
`random()`	renvoie un *décimal* aléatoire dans $[0;1[$
`uniform(a,b)`	choisit un *décimal* aléatoire dans $[a;b]$
`shuffle(seq)`	mélange la liste `seq` (en la modifiant)
`choice(seq)`	choisit un éléments aléatoirement dans la liste `seq`
`choices(seq,k=nb)`	tirage de `nb` éléments dans la liste `seq` avec répétition (on remet l'objet tiré avant de retirer)
`sample(seq,k=nb)`	tirage de `nb` (< `len(seq)`) éléments dans la liste `seq` sans répétition (comme si on tire `nb` éléments en même temps)

On peut afficher la description d'une fonction du module avec `help` :

```
>>> import random
>>> help(random.randrange)
Help on method randrange in module random:

randrange(start, stop=None, step=1) method of random.Random instance
    Choose a random item from range(stop) or range(start, stop[, step]).

    Roughly equivalent to ``choice(range(start, stop, step))`` but
    supports arbitrarily large ranges and is optimized for common cases.
```

Voici quelques exemples :

1. Quelques tirages :

```
>>> import random
>>> random.randrange(50,100,5)
90
>>> random.randint(50,100)
89
>>> random.choice([1,7,10,11,12,25])
25
>>> random.random()
0.20798545981630012
>>> random.uniform(10,20)
14.032980387401505
```

2. Simulons le lancé d'un dé :

```
>>> from random import *

>>> dico = {1:0, 2:0, 3:0, 4:0, 5:0, 6:0}
>>> # avec une compréhension :
>>> # dico = { c:0 for c in range(1,6)}

>>> for i in range(1000):
...     n = randint(1,6)
...     dico[n]+=1
...
>>> print(dico)
{1: 165, 2: 169, 3: 178, 4: 164, 5: 166, 6: 158}
```

3. On veut simuler 20 tirages du jeu "Pierre Feuille Ciseaux" sous forme d'une liste.

```
>>> from random import *
>>> seq = ["Pierre", "Feuille", "Ciseaux"]
>>> # print([choice(seq) for _ in range(20)])
>>> print(choices(seq,k=20))
['Feuille', 'Ciseaux', 'Ciseaux', 'Feuille', 'Ciseaux', 'Ciseaux', 'Pierre', 'Feuille', 'Ciseaux',
↪ 'Feuille', 'Pierre', 'Feuille', 'Pierre', 'Pierre', 'Feuille', 'Feuille', 'Ciseaux', 'Ciseaux',
↪ 'Feuille', 'Feuille']
```

Si vous exécutez vous-même les exemples précédents, vous devriez obtenir des résultats légèrement différents de ceux indiqués. C'est l'intérêt de l'aléatoire ! Cependant, pour des besoins de reproductibilité des analyses en science, on a souvent besoin de retrouver les mêmes résultats même si on utilise des nombres aléatoires. Pour cela, on peut définir ce qu'on appelle la "graine aléatoire". Cette graine se définit avec la fonction `seed()`

```
>>> import random
>>> random.seed(42)
>>> random.randint(0, 10)
10
>>> random.randint(0, 10)
1
>>> random.randint(0, 10)
0
```

Ici la graine aléatoire est fixée à 42. Si on ne précise pas la graine, par défaut Python utilise la date. Plus précisément, il s'agit du nombre de secondes écoulées depuis une date donnée du passé. Ainsi, à chaque fois qu'on relance Python, la graine sera différente car ce nombre de secondes sera différent. Si vous exécutez ces mêmes lignes de code, il se peut que vous ayez des résultats différents selon la version de Python. Néanmoins, vous devriez systématiquement obtenir les mêmes résultats si vous relancez plusieurs fois de suite ces instructions sur une même machine.

7.3.2. Le module `time` : accès à l'heure de l'ordinateur et aux fonctions gérant le temps

Le module `time` fournit beaucoup de fonctions en rapport avec le temps : gestions de calendriers, des systèmes horaires, d'horloges... Il permet aussi une estimation des temps d'exécution des programmes. La qualité de cette estimation dépend des mises en œuvre de ce module et des machines visées. [2]

La fonction `time()` est annoncée avec une précision de la seconde, sans que ce soit garanti pour toutes les machines. Une seconde est suffisamment long pour effectuer 2 giga = $2 \cdot 10^9$ opérations ! Elle peut donc être utile pour mesurer des simulations ou des exécutions conséquentes. La fonction `perf_counter()` permet les mesures les plus fines possibles adaptées à la mesure des temps d'exécution de programmes (sans être pour autant exactes, ni reproductibles). Elle utilise des fonctions spécifiques aux architectures des machines qui exploitent les compteurs matériels.

La fonction `perf_counter()` renvoie une durée de temps en secondes et permet de mesurer des durées d'exécution par soustraction. On procède en deux appels qui encadrent la portion de code à mesurer :

1. https://docs.python.org/fr/3/library/random.html
2. Mesurer le temps d'exécution d'un programme sur un ordinateur moderne est une tâche complexe. Il ne faut pas se laisser tromper par la rapidité apparente des résultats, car l'ordinateur gère de nombreuses tâches simultanément, telles que la gestion du système d'exploitation ou des périphériques. De plus, mesurer avec précision les temps d'exécution des programmes est difficile en raison de la structure complexe de la mémoire et des unités de calcul. En pratique, plus la durée que l'on souhaite mesurer est courte, plus la mesure sera sujette à l'incertitude et non reproductible. Pour obtenir des résultats significatifs, il est nécessaire de répéter l'exécution dans une boucle, mesurer le temps d'exécution de cette boucle et en calculer la moyenne. Les transferts de données entre la mémoire et les unités de calcul sont également un facteur crucial à prendre en compte, et leur durée peut être jusqu'à dix fois plus longue que celle des calculs proprement dits. À titre indicatif, il y a un facteur de 10 entre ce temps de transfert et le temps d'un calcul arithmétique élémentaire. Ainsi, les mesures peuvent être surprenantes lorsqu'elles sont influencées de manière significative par ces temps de transfert. Ces défis soulignent la différence entre les propriétés théoriques d'un algorithme et sa mise en œuvre pratique.

- `t0 = time.perf_counter()` : le premier appel initialise une valeur initiale `t0` ;
- `t = time.perf_counter()` : le second appel mesure `t` à l'issue de l'exécution de la portion de code,
- `t-t0` est la mesure de ce temps d'exécution (moyennant les réserves précédentes).

Voici un exemple qui permet de comparer le temps d'exécution pour ajouter des éléments à une liste avec trois techniques : l'opérateur `+`, la méthode `append` et la création par une liste en compréhension :

```
from time import perf_counter

N = 5*10**7

debut = perf_counter()
L = []
for i in range(N):
    L += [i]
fin = perf_counter()
print(f"Avec '+=' temps d'exécution = {fin-debut:g}s")

debut = perf_counter()
L = []
for i in range(N):
    L.append(i)
fin = perf_counter()
print(f"Avec 'append' temps d'exécution = {fin-debut:g}s")

debut = perf_counter()
L = [x for x in range(N)]
fin = perf_counter()
print(f"Avec une 'comprehensions-list' temps d'exécution = {fin-debut:g}s")
```

```
Avec '+=' temps d'exécution = 7.72694s
Avec 'append' temps d'exécution = 4.73519s
Avec une 'comprehensions-list' temps d'exécution = 2.43937s
```

On en conclut que l'utilisation des comprehensions-list est souvent à privilégier à l'utilisation de la méthode `append` pour construire des listes de manière efficace. Et `append` est plus efficace que l'instruction `+=`. On peut faire le même constant en comparant la méthode `extend` avec l'instruction `+=`. Cependant, cette règle n'est pas toujours absolue, car il existe des subtilités à considérer. Ces principes sont utiles, mais la réalité peut être un peu plus nuancée.

7.3.3. ☆ Le module `tabulate`

Le module `tabulate` offre une solution simple pour afficher des tableaux avec une mise en forme efficace. Il s'adapte au contenu des données, que ce soit des valeurs textuelles ou numériques, assurant un alignement intuitif des colonnes, la personnalisation des formats numériques, et même l'alignement par rapport au point décimal.

Le module fournit une seule fonction, `tabulate`, qui prend une liste de listes en entrée et génère un tableau bien structuré au format texte, prêt à être utilisé avec la fonction `print`. Voici un exemple de son fonctionnement de base :

```
from tabulate import tabulate
T = []
entete = [ "Nom" , "Age" ]
T.append([ "Alice" , 24 ])
T.append([ "Bob" , 9 ])
print ( tabulate (T, headers = entete ))
```

```
Nom      Age
-----  -----
Alice     24
Bob        9
```

Ce module offre de nombreuses options de personnalisation. Pour en savoir plus, consultez la documentation disponible à l'adresse suivante : https://pypi.org/project/tabulate/

7.3.4. ☆ Les modules `sys` et `os`

Le module `sys` contient des fonctions et des variables spécifiques à l'interpréteur Python lui-même. Par exemple, la fonction `sys.exit()` est utile pour quitter un script Python. On peut donner un argument à cette fonction (en général une

chaîne de caractères) qui sera renvoyé au moment où Python quittera le script. Par exemple, si vous attendez au moins un argument en ligne de commande, vous pouvez renvoyer un message pour indiquer à l'utilisateur ce que le script attend comme argument :

```
import sys

...

if a<0 :
    sys.exit(f"ERREUR car a<0: {a=}")
```

Le module `os` gère l'interface avec le système d'exploitation. La fonction `os.path.exists()` est une fonction pratique de ce module qui vérifie la présence d'un fichier sur le disque dur. Dans cet exemple, si le fichier n'existe pas sur le disque, on quitte le programme avec la fonction `exit()` du module `sys` que nous venons de voir :

```
import sys
import os

if os.path.exists("toto.pdb"):
    print("le fichier est présent")
else:
    sys.exit("le fichier est absent")
```

La fonction `os.getcwd()` renvoie le répertoire (sous forme de chemin complet) depuis lequel est lancé Python, ou encore la fonction `os.listdir()` renvoie le contenu de ce répertoire. Le résultat est renvoyé sous forme d'une liste contenant à la fois le nom des fichiers et des répertoires.

```
import os
print(os.getcwd())
print(f"Ce dossier contient {len(os.listdir())} fichiers et répertoires")
```

```
/home/minnolina/Dropbox/ENSEIGNEMENT/Cours/DSSC-1 Fondements de Python
Ce dossier contient 314 fichiers et répertoires
```

7.4. ☆ Modules personnels

Tout fichier source avec l'extension `.py` peut être utilisé comme un module. Supposons qu'une application python est composée de deux fichiers : `my_script.py` qui contient le programme principal et `my_module.py` qui regroupe les fonctions utilisées dans le script. Le nom du module correspond au nom du fichier. Le programme principal dans le script peut importer et utiliser les fonctions définies dans le module :

Fichier `my_module.py`

```
def f(x):
    return x**2+5

def afficher(x,y):
    print(f"Avec x={x} on obtient y={y}.")
```

Fichier `my_script.py`

```
import my_module as mm

x = 5
y = f(x)
afficher(x,y)
```

Avec x=5 on obtient y=30.

7.5. Exercices

Exercice 7.1 (Is the Water Boiling?)

L'eau entre en ébullition lorsque sa température dépasse la température de saturation. Cette température dépend de la pression et peut être estimée à l'aide de la loi d'Antoine, une relation empirique reliant pression et température :

$$T_{sat}(P) = \frac{B}{A - \log_{10}\left(\frac{P}{P_0}\right)} - C$$

où les coefficients pour l'eau sont

$$A = 3.55959, \quad B = 643.748, \quad C = -198.043, \quad P_0 = 100000\,\text{Pa}.$$

Les températures sont exprimées en kelvins et les pressions en pascals. Notons que P_0 est la pression atmosphérique, ainsi $T_{sat}(P_0) = \frac{B}{A} - C \approx 379$ K soit environ 100° C.

Objectif: écrire une fonction `is_boiling(T, P)` qui renvoie `True` si l'eau bout (*i.e.* si $T > T_{sat}$), et `False` sinon.

Exemples:

- `is_boiling(373, 101325)` ⇒ `True`
  À la pression atmosphérique, l'eau bout à 100 °C.
- `is_boiling(373, 50000)` ⇒ `False`
  À une pression plus faible, l'eau ne bout pas encore à 100 °C.
- `is_boiling(273, 10000)` ⇒ `True`
  À 10000 Pa, l'eau entre en ébullition vers 0 °C.

Correction

```
import math
A, B, C, P0 = 3.55959, 643.748, -198.043, 100000
Tsat = lambda pressure: B / (A - math.log10(pressure / P0)) - C
is_boiling = lambda T, p: T > Tsat(p)

# TESTS
from random import randint
for _ in range(5):
    T = randint(273, 373)       # Entre 0 et 100 degrés Celsius en Kelvin
    p = randint(10000, 50000)   # Entre 10 000 et 50 000 Pa
    print(f"Température = {T} K, Pression = {p} Pa => Sortie = {is_boiling(T, p)}")
```

```
Température = 331 K, Pression = 23696 Pa => Sortie = False
Température = 330 K, Pression = 44064 Pa => Sortie = False
Température = 370 K, Pression = 41132 Pa => Sortie = True
Température = 357 K, Pression = 17214 Pa => Sortie = True
Température = 351 K, Pression = 31603 Pa => Sortie = False
```

Exercice 7.2 (Benford's Law for the Second Digit)

La **loi de Benford** est une loi empirique qui décrit la distribution des premiers chiffres significatifs dans de nombreux ensembles de données réelles. Elle prédit que ces chiffres suivent une distribution logarithmique, et cette loi peut être généralisée au deuxième, troisième chiffre significatif, etc.

La fréquence relative du **deuxième chiffre significatif** $c \in \{0, 1, \ldots, 9\}$ (en base 10) prédite par la loi de Benford est donnée par

$$p(c) = \sum_{i=1}^{9} \log_{10}\left(1 + \frac{1}{10i + c}\right).$$

Objectif: écrire une fonction `p(c)` qui renvoie la probabilité $p(c)$ selon la formule ci-dessus.

- Le paramètre c est un entier qui représente un chiffre (de 0 à 9).
- Le résultat est une valeur de probabilité comprise entre 0 et 1.

Correction

```
from math import log10
p = lambda c : sum( [ log10( 1+1/(10*i+c)) for i in range( 1 , 10 ) ] )
```

```
# TESTS
for c in range(10):
    print(f"p({c})={p(c):.3f}")
```

```
p(0)=0.120          p(2)=0.109          p(4)=0.100          p(6)=0.093          p(8)=0.088
p(1)=0.114          p(3)=0.104          p(5)=0.097          p(7)=0.090          p(9)=0.085
```

Exercice 7.3 (Module `math` – π, e, $\sqrt{}$)

Considérons le triangle de cotés π^2, $\sqrt{\pi^5}$ et e^3. Est-il rectangle ? Utiliser le module `math`.

Correction

Pour déterminer si le triplet est pythagoricien, on commence par trier les trois longueurs par ordre croissant et on les assigne aux variables a, b, c. Ensuite, on vérifie si $a^2 + b^2 - c^2 \approx 0$. Étant donné l'importation d'un seul module, les quatre approches suivantes sont interchangeables, car il n'y a aucun risque de conflit avec d'autres modules.

Méthode 1:
```
import math
a, b, c = sorted([math.pi**2, math.sqrt(math.pi**5), math.e**3])
print( a**2+b**2-c**2 )
```
```
-1.7673451168320753e-05
```

Méthode 2:
```
import math as mm
a, b, c = sorted([mm.pi**2, mm.sqrt(mm.pi**5), mm.e**3])
print( a**2+b**2-c**2 )
```
```
-1.7673451168320753e-05
```

Méthode 3:
```
from math import pi, e, sqrt
a, b, c = sorted([pi**2, sqrt(pi**5), e**3])
print( a**2+b**2-c**2 )
```
```
-1.7673451168320753e-05
```

Méthode 4:
```
from math import *  # version du paresseux
a, b, c = sorted([pi**2, sqrt(pi**5), e**3])
print( a**2+b**2-c**2 )
```
```
-1.7673451168320753e-05
```

Au lieu d'utiliser la constante `math.e`, on aurait pu utiliser la fonction `math.exp` et donc écrire `math.exp(3)` au lieu de `math.e**3`.

Exercice 7.4 (Module `math` – sin, cos, tan, π, e)

Calculer avec le module `math` (resp. `numpy`) la valeur

$$\frac{\sin(3e^2)}{1+\tan\left(\frac{\pi}{8}\right)}.$$

Bonus : et avec le module `sympy` ?

Correction

Méthode 1:
```
import math
print( math.sin(3*math.e**2)/(1+math.tan(math.pi/8)) )
# ou
import numpy
print( numpy.sin(3*numpy.e**2)/(1+numpy.tan(numpy.pi/8)) )
```
```
-0.123823019762552
-0.123823019762552
```

Méthode 2:
```
import math as mm
print( mm.sin(3*mm.e**2)/(1+mm.tan(mm.pi/8)) )
# ou
import numpy as np
print( np.sin(3*np.e**2)/(1+np.tan(np.pi/8)) )
```

```
-0.123823019762552
-0.123823019762552
```

Méthode 3 :
```
from math import pi, e, sin, tan
print( sin(3*e**2)/(1+tan(pi/8)) )
# ou
from numpy import pi, e, sin, tan
print( sin(3*e**2)/(1+tan(pi/8)) )
```

```
-0.123823019762552
-0.123823019762552
```

Méthode 4 :
```
from math import *  # version du paresseux
print( sin(3*e**2)/(1+tan(pi/8)) )
# ou
from numpy import *  # version du paresseux
print( sin(3*e**2)/(1+tan(pi/8)) )
```

```
-0.123823019762552
-0.123823019762552
```

Au lieu d'utiliser la constante `math.e` (resp. `numpy.e`), on aurait pu utiliser la fonction `math.exp` (resp. `numpy.exp`) et donc écrire `math.exp(2)` (resp. `numpy.exp(2)`) au lieu de `math.e**2` (resp. `numpy.e**2`) .

Bonus : en utilisant la deuxième méthode (on peut bien-sûr utiliser aussi les autres méthodes)

```
import sympy as sym
val = sym.sin(3*sym.E**2)/(1+sym.tan(sym.pi/8))
print( val )
print( val.evalf() )
```

```
sqrt(2)*sin(3*exp(2))/2
-0.123823019762553
```

Exercice Bonus 7.5 ($\sqrt{\tan(\pi)}$)

Calculer avec python $\sqrt{\tan(\pi)}$. Que se passe-t-il ? Et avec le module `sympy` pour le calcul formel ?

Correction

On sait que $\tan(\pi) = 0$, mais `math.pi` est une approximation. La tangente de cette valeur approchée est négative, rendant la racine carrée non définie dans $\mathbb{R}$:

```
>>> from math import sqrt, tan, pi
>>> sqrt(tan(pi))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

Pour contourner cela, on peut utiliser le calcul formel avec `sympy` :

```
>>> from sympy import sqrt, tan, pi
>>> sqrt(tan(pi))
0
```

Source : https://scipython.com/book/chapter-9-general-scientific-programming/questions/evaluating-sqrttanpi/

Exercice 7.6 (Module `math` - Angles remarquables)

Soit $R = \left[0, \frac{\pi}{6}, \frac{\pi}{4}, \frac{\pi}{3}, \frac{\pi}{2}\right]$ une liste d'angles (en radians). Créer, par compréhension, une liste D avec les angles en degrés, une liste C avec l'évaluation du cosinus en ces angles et une liste S avec l'évaluation du sinus. Utiliser d'abord le module `math`, puis le module `numpy`.

Bonus : et avec le module `sympy` ?

Correction

1. Avec les module `math` et des listes en compréhension :

```
from math import pi, cos, sin # , degrees
R = [0,pi/6,pi/4,pi/3,pi/2]
D = [r*180/pi for r in R] # ou [degrees(r) for r in R]
C = [cos(r) for r in R]
S = [sin(r) for r in R]
print(f"{R = }")
print(f"{D = }")
print(f"{C = }")
print(f"{S = }")
```

```
R = [0, 0.5235987755982988, 0.7853981633974483, 1.0471975511965976, 1.5707963267948966]
D = [0.0, 29.999999999999996, 45.0, 59.99999999999999, 90.0]
C = [1.0, 0.8660254037844387, 0.7071067811865476, 0.5000000000000001, 6.123233995736766e-17]
S = [0.0, 0.49999999999999994, 0.7071067811865475, 0.8660254037844386, 1.0]
```

2. Avec les module numpy il n'est pas nécessaire d'utiliser des listes en compréhension car les fonctions sont vectorisées :

```
from numpy import array, pi, cos, sin
R = array([0,pi/6,pi/4,pi/3,pi/2])
D = R*180/pi # [r*180/pi for r in R]
C = cos(R) # [cos(r) for r in R]
S = sin(R) # [sin(r) for r in R]
print(f"{R = }")
print(f"{D = }")
print(f"{C = }")
print(f"{S = }")
```

```
R = array([0.        , 0.52359878, 0.78539816, 1.04719755, 1.57079633])
D = array([ 0., 30., 45., 60., 90.])
C = array([1.00000000e+00, 8.66025404e-01, 7.07106781e-01, 5.00000000e-01,
       6.12323400e-17])
S = array([0.        , 0.5       , 0.70710678, 0.8660254 , 1.        ])
```

3. Bonus (calcul formel) :

```
import sympy as sym
R = [0,sym.pi/6,sym.pi/4,sym.pi/3,sym.pi/2]
D = [r*180/sym.pi for r in R]
C = [sym.cos(r) for r in R]
S = [sym.sin(r) for r in R]
print(f"{R = }")
print(f"{D = }")
print(f"{C = }")
print(f"{S = }")
```

```
R = [0, pi/6, pi/4, pi/3, pi/2]
D = [0, 30, 45, 60, 90]
C = [1, sqrt(3)/2, sqrt(2)/2, 1/2, 0]
S = [0, 1/2, sqrt(2)/2, sqrt(3)/2, 1]
```

Exercice 7.7 (Module `math` - Paper folding)

Une feuille de papier d'une épaisseur d'un dixième de millimètre est pliée 15 fois en deux: quelle est l'épaisseur du résultat après pliage ? Après combien de pliages l'épaisseur dépasse-t-elle la distance Terre-Lune (la distance Terre-Lune vaut approximativement 384 400 km) ?

Pour calculer $\log_2(x)$ utiliser la fonction `log(x,2)` du module `math`.

Correction

Épaisseur du résultat après pliage : $2^{15} \times 0{,}1\,\text{mm} = 3{,}2768\,\text{m}$. Affichons les couples (`nb de pliages, épaisseur en mètre`) pour bien visualiser la croissance :

```
t = 1.e-4  # paper thickness, m
pliages = 19
print([(p,t*2**p) for p in range(pliages)])
```

```
[(0, 0.0001), (1, 0.0002), (2, 0.0004), (3, 0.0008), (4, 0.0016), (5, 0.0032), (6, 0.0064), (7, 0.0128),
↪ (8, 0.0256), (9, 0.0512), (10, 0.1024), (11, 0.2048), (12, 0.4096), (13, 0.8192), (14, 1.6384), (15,
↪ 3.2768), (16, 6.5536), (17, 13.1072), (18, 26.2144)]
```

L'épaisseur dépasse la distance Terre-Lune après n pliage avec n qui vérifie $2^n \times 10^{-4} \geq 3.844 \times 10^8$, *i.e.* $n \geq \log_2(3.844 \times 10^{12})$ ce qui correspond à 42 pliages.

```
from math import log
d = 384400 * 1.e3        # distance to moon, m
t = 1.e-4                # paper thickness, m
d_over_t = d/t
x = log(d_over_t, 2)     # base-2 logarithm
print(int(x)+1)
```

```
42
```

Exercice 7.8 (Module `math` – Approximation de la valeur ponctuelle de dérivées)
Écrire une fonction `derivatives` qui approche la valeur des dérivées première et seconde d'une fonction f en un point x par les formules

$$f'(x) \simeq \frac{f(x+h)-f(x-h)}{2h}, \qquad f''(x) \simeq \frac{f(x+h)-2f(x)+f(x-h)}{h^2}.$$

Pour la fonction $x \mapsto \cos(x)$ en le point $x = \frac{\pi}{2}$, comparer la valeur exacte avec la valeur approchée par ces deux expressions (à vous de choisir une valeur de h satisfaisante).

Correction

```
def derivatives(f, x, h):
    d1f = (f(x+h)-f(x-h))/(2*h)
    d2f = (f(x+h)-2*f(x)+f(x-h))/h**2
    return d1f, d2f
```

Si $f(x) = \cos(x)$ alors $f'(x) = -\sin(x)$ et $f''(x) = -\cos(x)$ donc $f'\left(\frac{\pi}{2}\right) = 1$ et $f''\left(\frac{\pi}{2}\right) = 0$. Est-ce que les valeurs calculées par notre fonction python sont précises ? Cela dépend de h bien-sûr. Pour $h = 10^{-5}$ on a :

```
import math
d1f, d2f = derivatives( f=math.cos, x=math.pi/2, h=1e-5 )
print(f"f' (pi/2) ≈ {d1f}")
print(f"f''(pi/2) ≈ {d2f}")
```

```
f' (pi/2) ≈ -0.9999999999898845
f''(pi/2) ≈ 1.6940658945086004e-11
```

Exercice 7.9 (Module `math` – Factorielle)
La factorielle d'un nombre entier n, notée $n!$, est définie comme le produit $n \times (n-1) \times \cdots \times 3 \times 2 \times 1$. Par convention $0! = 1$. Par exemple, $10! = 10 \times 9 \times \cdots \times 3 \times 2 \times 1 = 3628800$.

Écrivez une fonction `my_factorial` qui calcule la factorielle d'un entier naturel n en utilisant les produits. Ensuite, pour tester la fonction, vous comparez le résultat obtenu par votre fonction à celui de la fonction `factorial` du module `math`.

Correction

- Version non récursive : elle utilise une boucle pour calculer la factorielle. Elle est plus efficace en termes d'utilisation de la pile mémoire et recommandée pour des valeurs élevées de n.

```
#def my_factorial(n):
#    x=1
#    for i in range(2,n+1):
#        x*=i
#    return x

from math import prod # Nécessite Python > 3.8
my_factorial = lambda n : prod([i for i in range(2,n+1)])

# TESTS
from math import factorial # Import de la fonction de référence
for n in [0,2,3,6,10,50]:
    print(factorial(n)==my_factorial(n), end=" ")
```

```
True True True True True True
```

- Version récursive : elle s'appuie directement sur la propriété $n! = n \times (n-1)!$. Elle est plus intuitive et naturelle pour représenter des définitions mathématiques, mais peut provoquer une `RecursionError` pour des valeurs très grandes de n.

```
my_factorial = lambda n : n*my_factorial(n-1) if n>1 else 1

# TESTS
from math import factorial  # Import de la fonction de référence
for n in [0,2,3,6,10,50]:
    print(factorial(n)==my_factorial(n), end=" ")
```

```
True True True True True True
```

Exercice 7.10 (Module `math` – Factorion)

Un factorion est un nombre entier naturel qui est égal à la somme des factorielles de ses chiffres. Trouver les nombre factorions inférieurs à 10^6 (pour calculer la factorielle d'un nombre on utilisera la fonction `factorial` du module `math`).

Correction

```
from math import factorial

liste_factorions = []
for n in range(10**6+1):
    if n == sum( factorial(int(i)) for i in str(n) ):
        liste_factorions.append(n)

print(f"Les factorions inférieurs à 10^6 sont {liste_factorions}")
```

```
Les factorions inférieurs à 10^6 sont [1, 2, 145, 40585]
```

Exercice 7.11 (Module `math` – Stirling)

La formule de Stirling, du nom du mathématicien écossais James Stirling, donne un équivalent de la factorielle d'un entier naturel n quand n tend vers l'infini :

$$\ln(n!) \approx n\ln(n) - n.$$

1. Créez une fonction `exacte(n)` qui prend en entrée un entier n et renvoie $\ln(n!)$, soit en s'appuyant sur la fonction `factorial` du module `math`, soit en remarquant que le logarithme d'un produit est la somme des logarithmes des facteurs.
2. Créez une fonction `approche(n)` qui prend en entrée un entier n et renvoie $n\ln(n) - n$.
3. Écrire un script qui calcule la plus petite valeur de n telle que l'erreur relative
   `|exacte(n)-approche(n)|/exacte(n)`
   est strictement inférieure à 1%.

Source : https://scipython.com/book/chapter-2-the-core-python-language-i/questions/problems/p24/stirlings-approximation/

Correction

La valeur exacte $\ln(n!)$, le logarithme naturel de la factorielle de n, peut être calculée directement ou en utilisant la propriété du logarithme d'un produit. En effet, nous avons $\ln(n!) = \ln(n \times (n-1) \times \cdots \times 2 \times 1) = \ln(n) + \ln(n-1) + \cdots + \ln(2) + \ln(1)$. Ce calcul peut également être exprimé sous forme compacte à l'aide d'une somme

$$\ln(n!) = \ln\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \ln(i).$$

```
from math import log, factorial

exacte    = lambda n : log(factorial(n))
exactebis = lambda n : sum(log(i) for i in range(1,n+1))
approche  = lambda n : n*log(n)-n
```

```
def erreur(n):
    ex = exacte(n)
    ap = approche(n)
    return abs(ex-ap)/ex

# main
n = 2
while erreur(n)>=0.01:
    n+=1

print(f"{n = }, {exacte(n) = :g}, {exactebis(n) = :g}, {approche(n) = :g}, {erreur(n) = :g}")
```

```
n = 90, exacte(n) = 318.153, exactebis(n) = 318.153, approche(n) = 314.983, erreur(n) = 0.00996305
```

Exercice 7.12 (Détermination d'un seuil, suite de type ①)

Soit $(u_n)_{n\in\mathbb{N}}$ la suite définie par

$$u_n = \frac{n}{\sqrt[n]{n!}}.$$

On peut montrer que cette suite tend vers e lorsque n tend vers $+\infty$. Cela signifie que

$$\forall \varepsilon > 0 \quad \exists n \in \mathbb{N} \quad : \quad \forall n \geq \mathrm{N} \quad |u_n - e| \leq \varepsilon.$$

Écrivez le code de la fonction `seuil` qui prend en paramètre le nombre `eps` et renvoie la valeur du plus petit entier n tel que l'on ait $|u_n - e| \leq \varepsilon$.

Remarque : au chapitre 4 on a vu différents types de suite. Il s'agit d'une suite de type ①.

Correction

```
from math import factorial, e

def seuil(eps):
    u = lambda n : n / factorial(n)**(1/n)
    n = 1
    while abs( u(n) - e) > eps:
        n += 1
    return n

# TESTS
for eps in [ 1, 0.1, 0.06 ]:
    print(f"Le plus petit n tel que |u_n-e|<={eps} est {seuil(eps)}.")
```

```
Le plus petit n tel que |u_n-e|<=1 est 4.
Le plus petit n tel que |u_n-e|<=0.1 est 84.
Le plus petit n tel que |u_n-e|<=0.06 est 155.
```

Exercice 7.13 (Module `math` – Projet Euler n°20 et Défi Turing n°6 – somme de chiffres)

$10! = 3628800$ et la somme de ses chiffres vaut $3+6+2+8+8+0+0 = 27$. Trouver la somme des chiffres du nombre $100!$ (projet Euler) puis la somme des chiffres du nombre $2013!$ (défi Turing).

Correction

Pour résoudre ce problème, la première étape consiste à calculer la valeur de $100!$ en utilisant, par exemple, la fonction `factorial` du module `math`. Ensuite, on convertit le nombre obtenu en une chaîne de caractères à l'aide de la fonction `str`. On procède ensuite à la lecture des chiffres un par un, puis on les transforme en entiers en utilisant la fonction `int`. Enfin, on les additionne.

```
import math
somme_chiffres_fact = lambda n : sum([int(x) for x in str(math.factorial(n))])

# TESTS
print(somme_chiffres_fact(10))
print(somme_chiffres_fact(100))
# print(somme_chiffres_fact(2013))
# ValueError: Exceeds the limit (4300 digits) for integer string conversion
```

```
27
648
```

Pour résoudre le défis Turing ce n'est pas aussi simple. Pour 2013!, nous devons adopter une autre approche car on ne peut pas manipuler une chaîne de caractères aussi grande. Une solution consiste à utiliser une division successive par 10 pour extraire les chiffres du nombre, ce qui permet de les additionner directement sans conversion en chaîne.

```
import math
def somme_chiffres_fact(n):
    fact = math.factorial(n)
    somme = 0
    while fact > 0:
        fact,r = divmod(fact,10)
        somme += r
    return somme

# TESTS
print(somme_chiffres_fact(10))
print(somme_chiffres_fact(100))
print(somme_chiffres_fact(2013))
```

```
27
648
24021
```

Exercice 7.14 (Module `math` – Nombres de Brown)
Les nombres de Brown sont des couples d'entiers m et n tels que $m^2 = n! + 1$. On peut montrer qu'ils sont tous inférieurs à 100. Calculer tous les nombres de Brown.

Correction

```
>>> from math import factorial
>>> print([ (m,n) for m in range(100) for n in range(100) if m**2==factorial(n)+1 ])
[(5, 4), (11, 5), (71, 7)]
```

Exercice 7.15 (Module `math` – Approximation de e)
Notons $f(i) = \frac{1}{i!}$ et $\tilde{e}(n) = \sum_{i=0}^{n} f(i)$. On sait que

$$e = \lim_{n\to\infty} \tilde{e}(n).$$

1. Créez un fonction `f(i)` qui prend en entrée un entier i et renvoie la valeur $f(i) = \frac{1}{i!}$ en utilisant la fonction `factorial` du module `math`.
2. Créez ensuite une fonction `approx_e(n)` qui prend en entrée un entier n et renvoie $\tilde{e}(n) = \sum_{i=0}^{n} f(i)$ (on construit la liste `[f(0), f(1), ..., f(n)]` en compréhension et on somme ses termes).
3. Écrire un script qui calcule la plus petite valeur de n telle que `approx_e(n) = math.e` à 10^{-15} près.

Correction

```
import math
f = lambda x : 1/math.factorial(x)
approx_e = lambda n : sum([f(i) for i in range(n+1) ])

print(f"Avec n = {10} on obtient e ≈ {approx_e(10)}")

# Méthode non optimisée
n = 0
while abs(approx_e(n)-math.e)>1e-15 and n<100:
    n+=1
print(f"Avec n = {n} on obtient\n{approx_e(n)=}\n{     math.e=}")

# Mieux: on ne recalcule pas toute la liste mais on ajoute juste un terme à chaque itération
n = 0
```

```
e_approx = f(n)
while abs(e_approx-math.e)>1e-15 and n<100:
    n += 1
    e_approx +=f(n)
print(f"Avec n = {n} on obtient\n{approx_e(n)=}\n{     math.e=}")
```

```
Avec n = 10 on obtient e ≈ 2.7182818011463845
Avec n = 17 on obtient
approx_e(n)=2.718281828459045
     math.e=2.718281828459045
Avec n = 17 on obtient
approx_e(n)=2.718281828459045
     math.e=2.718281828459045
```

Exercice 7.16 (Modules `math` et `Decimal` – Polignac)
Vérifier la formule de Polignac pour le calcul du nombre de zéros à la fin de $n!$:

$$n! \text{ se termine par } \sum_i \left\lfloor \frac{n}{5^i} \right\rfloor \text{ zéros.}$$

Par exemple :
- $9! = 362880$ se termine par 1 zéro,
- $10! = 3628800$ se termine par 2 zéros,
- $20! = 2432902008176640000$ se termine par 4 zéros.

Source : https://scipython.com/book/chapter-2-the-core-python-language-i/questions/problems/p25/de-polignacs-formula/

Correction
Méthode 1 : méthode analytique avec division par des puissances de 5

```
from math import factorial # Pour verification

for n in [9,10,20]:
    nzeros = 0
    term = n
    while True:
        term //= 5
        if term == 0:
            break
        nzeros += term
    print(f'{n:d}! se termine par {nzeros:d} zéros
    ↪  : {n:d}! = {factorial(n):d}')
```

```
9! se termine par 1 zéros : 9! = 362880
10! se termine par 2 zéros : 10! = 3628800
20! se termine par 4 zéros : 20! =
↪  2432902008176640000
```

Méthode 2 : calcule explicite de la factorielle puis conversion `int` → `str`

```
from math import factorial

for n in [9,10,20]:
    nf = factorial(n)
    # Méthode 2.1
#   nzeros = 0
# for c in str(nf)[::-1]:
#     if c != '0':
#         break
#     nzeros += 1
    # Méthode 2.2
    s = str(nf)
    nzeros = len(s)-len(s.strip('0'))
    print(f'{n:d}! se termine par {nzeros:d} zéros
    ↪  : {n:d}! = {nf:d}')
```

```
9! se termine par 1 zéros : 9! = 362880
10! se termine par 2 zéros : 10! = 3628800
20! se termine par 4 zéros : 20! =
↪  2432902008176640000
```

Remarque : pour des valeurs très grandes de n, la conversion de `int` vers `str` peut poser problème à cause de la taille de l'entier. En effet, il y a une limite explicite sur la taille maximale des entiers convertis en chaîne de caractères :

```
>>> import sys
>>> sys.get_int_max_str_digits()
4300
```

Si $n!$ ne tient pas dans 4300 chiffres, `str(factorial(n))` lève une erreur (sauf si on augmente la limite). Or 1558! tient sur 4300 chiffres, mais 1559! no. On peut contourner cela en utilisant le module `Decimal` pour manipuler les très grands entiers :

```
from math import factorial
from decimal import Decimal

for n in [9,10,20]:
    nf = factorial(n)
    s = str(Decimal(nf))
    nzeros = len(s) - len(s.strip('0'))
    print(f'{n}! se termine par {nzeros} zéro(s) : {n}! = {s}')
```

```
9! se termine par 1 zéro(s) : 9! = 362880
10! se termine par 2 zéro(s) : 10! = 3628800
20! se termine par 4 zéro(s) : 20! = 2432902008176640000
```

Exercice 7.17 (Suites type ③: comparaison itératif / formule explicite)

On considère une suite (u_n) définie par récurrence linéaire d'ordre 2 :

$$u_{n+1} = \alpha u_n + \beta u_{n-1}, \qquad u_0, u_1 \in \mathbb{R}.$$

On peut donner une expression explicite du terme u_n ; considérons l'équation caractéristique

$$r^2 - \alpha r - \beta = 0, \qquad \Delta = \alpha^2 + 4\beta.$$

Trois cas possibles selon Δ :

- $\Delta > 0$: deux racines réelles distinctes $r_{1,2} = \frac{\alpha \pm \sqrt{\Delta}}{2}$ et $u_n = \mathrm{A} r_1^n + \mathrm{B} r_2^n$
- $\Delta = 0$: racine double $r = \frac{\alpha}{2}$ et $u_n = (\mathrm{A}n + \mathrm{B}) r^n$
- $\Delta < 0$: racines complexes $r_{1,2} = \frac{\alpha}{2} \pm i \frac{\sqrt{-\Delta}}{2}$ et $u_n = \rho^n \big(\mathrm{C}\cos(n\theta) + \mathrm{D}\sin(n\theta)\big)$ avec $\rho = \sqrt{-\beta}$, et $\cos(\theta) = \frac{\alpha}{2\rho}$.

Les constantes $\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}$ sont déterminées par les conditions initiales u_0, u_1 :

- $\Delta > 0$: $\begin{cases} u_0 = \mathrm{A} + \mathrm{B} \\ u_1 = \mathrm{A} r_1 + \mathrm{B} r_2 \end{cases} \rightsquigarrow \mathrm{A} = \frac{u_1 - r_2 u_0}{r_1 - r_2}$ et $\mathrm{B} = u_0 - \mathrm{A}$;
- $\Delta = 0$: $\begin{cases} u_0 = \mathrm{B} \\ u_1 = (\mathrm{A} + \mathrm{B}) r \end{cases} \rightsquigarrow \mathrm{B} = u_0$ et $\mathrm{A} = \frac{u_1 - r u_0}{r}$;
- $\Delta < 0$: $\begin{cases} u_0 = \mathrm{C} \\ u_1 = \rho(\mathrm{C}\cos(\theta) + \mathrm{D}\sin(\theta)) \end{cases} \rightsquigarrow \mathrm{C} = u_0$ et $\mathrm{D} = \frac{u_1/\rho - \mathrm{C}\cos(\theta)}{\sin(\theta)}$.

Par exemple, la suite de Fibonacci correspond à $\alpha = \beta = 1$, $u_0 = 0$, $u_1 = 1$. L'équation caractéristique est $x^2 - x - 1 = 0$, donc $\Delta = 5 > 0$. Les racines sont $r_{1,2} = \frac{1 \pm \sqrt{5}}{2}$. La forme générale est ainsi $u_n = \mathrm{A} r_1^n + \mathrm{B} r_2^n$ avec $\mathrm{A} + \mathrm{B} = 0$, $\mathrm{A} r_1 + \mathrm{B} r_2 = 1$, d'où la formule de Binet

$$u_n = \frac{1}{\sqrt{5}} \left(r_1^n - r_2^n \right).$$

Exercice : Écrire une fonction `recurrence(alpha,beta,u0,u1,N)` qui calcule u_{N} de deux façons : par itération directe et par la formule explicite utilisant les racines de l'équation caractéristique. Vérifier la fonction sur trois cas :

1. Δ>0 : suite de Fibonacci ($\alpha = \beta = 1, u_0 = 0, u_1 = 1$)
2. Δ=0 : $u_{n+1} = 2u_n - u_{n-1}$, $u_0 = 1, u_1 = 2$
3. Δ<0 : $u_{n+1} = -u_{n-1}$, $u_0 = 0, u_1 = 1$

Correction

```
import math

def recurrence_list(alpha, beta, u0, u1, N):

    # ---- Méthode itérative ----
    uu_iter = [u0, u1] if N >= 1 else [u0]
    for n in range(2, N+1):
        uu_iter.append(alpha*uu_iter[-1] + beta*uu_iter[-2])
```

```python
    # ---- Méthode exacte ----
    delta = alpha**2 + 4*beta
    uu_exact = []

    if delta > 0:
        r1 = (alpha + math.sqrt(delta)) / 2
        r2 = (alpha - math.sqrt(delta)) / 2
        A = (u1 - r2*u0)/(r1 - r2)
        B = u0 - A
        uu_exact = [A*(r1**n) + B*(r2**n) for n in range(N+1)]

    elif delta == 0:
        r = alpha/2
        Bc = u0
        A = (u1 - r*Bc)/r if r != 0 else 0
        uu_exact = [(A*n + Bc)*(r**n) for n in range(N+1)]

    else:  # Δ<0, racines complexes via trig
        rho = math.sqrt(-beta)
        theta = math.acos(alpha/(2*rho))
        C = u0
        D = (u1 - C*math.cos(theta))/math.sin(theta)
        uu_exact = [(rho**n)*(C*math.cos(n*theta) + D*math.sin(n*theta)) for n in range(N+1)]

    return uu_iter, uu_exact

# -------------------------
# Tests
N = 5

# Δ>0 : Fibonacci
fib_iter, fib_exact = recurrence_list(1, 1, 0, 1, N)
print("Δ>0 :")
print("Itératif :", fib_iter)
print("Exact    :", fib_exact)
print()

# Δ=0
d0_iter, d0_exact = recurrence_list(2, -1, 1, 2, N)
print("Δ=0 :")
print("Itératif :", d0_iter)
print("Exact    :", d0_exact)
print()

# Δ<0
dneg_iter, dneg_exact = recurrence_list(0, -1, 0, 1, N)
print("Δ<0 :")
print("Itératif :", dneg_iter)
print("Exact    :", dneg_exact)
```

```
Δ>0 :
Itératif : [0, 1, 1, 2, 3, 5]
Exact    : [0.0, 1.0, 0.9999999999999999, 2.0, 3.0000000000000004, 5.000000000000001]

Δ=0 :
Itératif : [1, 2, 3, 4, 5, 6]
Exact    : [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]

Δ<0 :
Itératif : [0, 1, 0, -1, 0, 1]
Exact    : [0.0, 1.0, 1.2246467991473532e-16, -1.0, -2.4492935982947064e-16, 1.0]
```

Dans chaque cas, on constate que les deux méthodes donnent le même résultat mathématique, mais le type est différent.

Exercice 7.18 (Module `math` – Indicatrice d'Euler)

L'indicatrice d'Euler est une fonction qui, à tout entier naturel n non nul, associe le nombre d'entiers compris entre 1 et n (inclus) qui sont premiers avec n :

$$\begin{aligned}\varphi\colon \mathbb{N}^* &\to \mathbb{N}^*\\ n &\mapsto \operatorname{card}\left\{ m \in \mathbb{N}^* \;\middle|\; m \leq n \text{ et } \gcd(m,n) = 1 \right\}.\end{aligned}$$

Elle est aussi appelée fonction phi d'Euler ou simplement fonction phi, car la lettre φ (ou ϕ) est communément utilisée pour la désigner.

Exemples :

- $\varphi(8) = 4$ car parmi les nombres de 1 à 8, seuls les quatre nombres 1, 3, 5 et 7 sont premiers avec 8 ;
- $\varphi(12) = 4$ car parmi les nombres de 1 à 12, seuls les quatre nombres 1, 5, 7 et 11 sont premiers avec 12 ;
- $\varphi(1) = 1$ car 1 est premier avec lui-même (c'est le seul entier naturel qui vérifie cette propriété).

Notons qu'un entier $p > 1$ est premier si et seulement si tous les nombres de 1 à $p-1$ sont premiers avec p, c.-à-d. si et seulement si $\varphi(p) = p - 1$.

Écrire une fonction qui, pour $n \in \mathbb{N}$, renvoie $\varphi(n)$. On utilisera la fonction `gcd` du module `math`.

Correction

```python
from math import gcd
phi = lambda n: sum(gcd(n, k) == 1 for k in range(1, n+1))
print(*[f'phi({n}) = {phi(n)}' for n in range(1, 13)], sep=", ", end=". ")
```

```
phi(1) = 1, phi(2) = 1, phi(3) = 2, phi(4) = 2, phi(5) = 4, phi(6) = 2, phi(7) = 6, phi(8) = 4, phi(9) =
↪  6, phi(10) = 4, phi(11) = 10, phi(12) = 4.
```

Exercice 7.19 (Module `math` – Énoncer des chiffres)

Transforme un nombre n en une chaîne de caractères, sélectionner les x premières décimales et énoncer les chiffres (en français). Par exemple, si $n = \pi$ et $x = 2$, il devra afficher `trois virgule un quatre`. Pour simplifier, nous considérons que des nombres positifs inférieurs à 10.

Source : https://scipython.com/book/chapter-2-the-core-python-language-i/questions/problems/p24/pi-read-aloud/

Correction

```python
def enoncer(n,x):
    num = ('zero', 'un', 'deux', 'trois', 'quatre', 'cinq', 'six', 'sept', 'huit', 'neuf')
    print(f"{n} approché par {x} chiffres après la virgule vaut {round(n,x)}")
    n = str(n)
    print(f"{num[int(n[0])]} virgule ", end='')
    for i in n[2:x+2]:
        print(num[int(i)], end=' ')
    print()

import math
enoncer(math.pi,2)
enoncer(math.e,4)
```

```
3.141592653589793 approché par 2 chiffres après la virgule vaut 3.14
trois virgule un quatre
2.718281828459045 approché par 4 chiffres après la virgule vaut 2.7183
deux virgule sept un huit deux
```

Exercice 7.20 (Module `math` – Overflow)

Calculer l'hypoténuse d'un triangle rectangle dont les cotés mesurent 1.5×10^{200} et 3.5×10^{201}.

Correction

Calcul naif :

```
>>> from math import sqrt
>>> a,b = 1.5e200, 3.5e201
>>> sqrt(a**2+b**2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Numerical result out of range')
```

Le problème est que, bien que les valeurs numériques des longueurs des deux côtés puissent toutes être représentées par des float (*i.e.* ils ne sont pas trop grand), le calcul de la longueur de l'hypoténuse sous forme de $c = \sqrt{a^2+b^2}$ nécessite la somme de carrés qui eux débordent. Deux stratégies possibles :

- on remarque que $c = b\sqrt{1+\left(\frac{a}{b}\right)^2}$; le carré ne peut pas déborder si $b > a$, sinon on factorise a :
  ```
  >>> from math import sqrt
  >>> a,b = 1.5e200, 3.5e201
  >>> b*sqrt(1+(a/b)**2)
  3.5032128111206724e+201
  ```
- on peut utiliser la fonction `hypot`
  ```
  >>> from math import hypot
  >>> a,b = 1.5e200, 3.5e201
  >>> hypot(a,b)
  3.503212811120672e+201
  ```

Bien sûr, on peut aussi commencer par simplifier les calculs sur papier :

$$c = \sqrt{a^2+b^2} = \sqrt{\left(\frac{3}{2}10^{200}\right)^2 + \left(\frac{7}{2}10^{201}\right)^2} = \sqrt{\frac{9}{4}10^{400} + \frac{49}{4}10^{402}} = \sqrt{\left(\frac{9}{4}+\frac{49}{4}10^2\right)10^{400}} = 10^{200} \times \sqrt{\frac{9}{4}+\frac{49}{4}10^2}$$

```
>>> from math import sqrt
>>> 10**200 * sqrt( 9/4 + 49*100/4 )
3.503212811120672e+201
```

Exercice 7.21 (Module `math` – Distance)

① Soit `P=[Px,Py]` la liste contenant les coordonnées d'un point du plan $P = (P_x, P_y)$. On définit la distance entre deux points P et Q par

$$d(P,Q) = \sqrt{(P_x - Q_x)^2 + (P_y - Q_y)^2}.$$

Écrire une fonction `distance(P,Q)` qui retourne $d(P,Q)$.

Exemple : Si P = [0, 0] et Q = [0, 1] alors distance(P,Q) = 1.0

② Soit P un point et L une liste de points. Écrire une fonction `plus_proche(P,L)` qui renvoie le point de la liste L se trouvant à la plus courte distance du point P.

③ Soit `T=[P,Q,R]` une liste contenant trois points du plan.

- Écrire une fonction `perimetre(T)` qui retourne le périmètre du triangle de sommets P, Q et R (en utilisant la fonction `distance`).
- Écrire une fonction `IsEquilateral(T)` qui retourne `True` si le triangle est équilatère, `False` sinon.

Exemple : Si T = [[0, 0], [0, 1], [1, 0]] alors perimetre(T) = 3.41421 et IsEquilateral(T) = False

④ Soit `C=[P,r]` une liste contenant un point du plan et une valeur positive, représentant un cercle de centre P et rayon r.

- Écrire une fonction `superpose(C1,C2)` qui retourne `True` si les deux cercles C1 et C2 se chevauchent, `False` sinon.

Correction

① Distance entre deux points

```
import math
distance = lambda P,Q : math.sqrt((P[0]-Q[0])**2+(P[1]-Q[1])**2)
# distance = lambda P,Q : math.sqrt( sum(  (p-q)**2 for p,q in zip(P,Q)  )  )
# distance = lambda P,Q : math.dist(P,Q) # si Python > 3.8

# TEST
P, Q = [0,0] , [0,1]
print(f"{distance(P,Q) = }")
```

```
distance(P,Q) = 1.0
```

② Point d'une liste le plus proche d'un point donné

```
plus_proche = lambda P,L : min( L , key=lambda ell:distance(ell,P) )

#def plus_proche(P, L):
#    point_plus_proche, dist = L[0], distance(L[0],P)
#    for point in L[1:]:
#        new_dist = distance(point, P)
#        if new_dist<dist:
#            dist = new_dist
#            point_plus_proche = point
#    return point_plus_proche

# TEST
P = [0,0]
L = [ [1,0] , [0,1] , [1,1] , [-0.5,-0.5]   ]
N = plus_proche(P,L)
print(f"plus_proche(P,L) renvoie {N}, en effet la distance vaut {distance(P,N)}")
```

```
plus_proche(P,L) renvoie [-0.5, -0.5], en effet la distance vaut 0.7071067811865476
```

③ Triangles

```
perimetre = lambda T : distance(T[0],T[1])+distance(T[0],T[2])+distance(T[1],T[2])
# perimetre = lambda T : sum( [ distance(T[i%3],T[(i+1)%3]) for i in range(3) ] )
IsEquilateral = lambda T: abs(distance(T[0],T[1])-distance(T[0],T[2]))<1.e-10 and \
                                  abs(distance(T[0],T[1])-distance(T[1],T[2]))<1.e-10

# TEST 1
P, Q, R = [0,0] , [0,1] , [1,0]
T = [P,Q,R]
print(f"{T = }")
print(f"{perimetre(T) = }")
print(f"{IsEquilateral(T) = }")

# TEST 2
P, Q, R = [0,0] , [1,0] , [0.5,math.sqrt(1-0.25)]
T = [P,Q,R]
print(f"{T = }")
print(f"{perimetre(T) = }")
print(f"{IsEquilateral(T) = }")
```

```
T = [[0, 0], [0, 1], [1, 0]]
perimetre(T) = 3.414213562373095
IsEquilateral(T) = False
T = [[0, 0], [1, 0], [0.5, 0.8660254037844386]]
perimetre(T) = 3.0
IsEquilateral(T) = True
```

Remarque : lorsqu'il s'agit juste de comparer deux distances, il suffit de comparer les carrés (en évitant l'extraction de la racine carré).

```
d2 = lambda P,Q : (P[0]-Q[0])**2+(P[1]-Q[1])**2
IsEquilateral = lambda T: abs(d2(T[0],T[1])-d2(T[0],T[2]))<1.e-10 and
↪  abs(d2(T[0],T[1])-d2(T[1],T[2]))<1.e-10

# TEST 1
P, Q, R = [0,0] , [0,1] , [1,0]
T = [P,Q,R]
print(f"{T = }")
print(f"{IsEquilateral(T) = }")

# TEST 2
P, Q, R = [0,0] , [1,0] , [0.5,(1-0.25)**0.5]
T = [P,Q,R]
print(f"{T = }")
print(f"{IsEquilateral(T) = }")
```

```
T = [[0, 0], [0, 1], [1, 0]]
IsEquilateral(T) = False
T = [[0, 0], [1, 0], [0.5, 0.8660254037844386]]
IsEquilateral(T) = True
```

④ Cercles.
Pour savoir si deux cercles se chevauchent, il suffit de vérifier que la distance entre les deux centres est inférieure à la somme de leurs rayons. On peut à nouveau éviter le calcul de la racine carrée : $d(\mathrm{P}_1,\mathrm{P}_2) \leq r_1 + r_2$ ssi $(d(\mathrm{P}_1,\mathrm{P}_2))^2 \leq (r_1+r_2)^2$:

```
d2 = lambda P,Q : (P[0]-Q[0])**2+(P[1]-Q[1])**2
superpose = lambda C1,C2 : d2(C1[0],C2[0]) <= (C1[1]+C2[1])**2

# TEST 1
C1, C2 = [[0,0],1] , [[1,1],1]
print(f"Cercle 1: centre ({C1[0][0]},{C1[0][1]}), rayon {C1[1]}")
print(f"Cercle 2: centre ({C2[0][0]},{C2[0][1]}), rayon {C2[1]}")
print(f"Carré de la distance des deux centres: {d2(C1[0],C2[0])}")
print(f"Carré de la somme des deux rayons: {(C1[1]+C2[1])**2}")
print(f"{superpose(C1,C2) = }")

# TEST 2
C1, C2 = [[0,0],1] , [[3,0],1]
print(f"Cercle 1: centre ({C1[0][0]},{C1[0][1]}), rayon {C1[1]}")
print(f"Cercle 2: centre ({C2[0][0]},{C2[0][1]}), rayon {C2[1]}")
print(f"Carré de la distance des deux centres: {d2(C1[0],C2[0])}")
print(f"Carré de la somme des deux rayons: {(C1[1]+C2[1])**2}")
print(f"{superpose(C1,C2) = }")
```

```
Cercle 1: centre (0,0), rayon 1
Cercle 2: centre (1,1), rayon 1
Carré de la distance des deux centres: 2
Carré de la somme des deux rayons: 4
superpose(C1,C2) = True
Cercle 1: centre (0,0), rayon 1
Cercle 2: centre (3,0), rayon 1
Carré de la distance des deux centres: 9
Carré de la somme des deux rayons: 4
superpose(C1,C2) = False
```

Exercice 7.22 (Module `math` – Distance point droite)

Considérons trois points V, W et P dans un plan cartésien. Écrivez une fonction `dist_point_droite` qui prend en entrée trois listes représentant les coordonnées des points V, W, et P, et qui renvoie la distance de P par rapport à la droite passant par les points V et W.

Rappel : la distance du point $\mathrm{P} = (x_p, y_p)$ à la droite r d'équation $ax + by + c = 0$ est donnée par

$$d(\mathrm{P}, r) = \frac{|ax_p + by_p + c|}{\sqrt{a^2+b^2}}.$$

Aide pour déterminer l'équation de la droite passant par les points $\mathrm{V} = (x_v, y_v)$ et $\mathrm{W} = (x_w, y_w)$:

- Si $x_v = x_w$, la droite est verticale avec pour équation $x = x_v$. On peut choisir $a = 1$, $b = 0$ et $c = -x_v$.
- Si $x_v \neq x_w$, la droite a pour équation $y = \frac{y_w - y_v}{x_w - x_v}(x - x_v) + y_v$. Dans ce cas, on peut choisir $a = \frac{y_w - y_v}{x_w - x_v}$, $b = -1$ et $c = y_v - \frac{y_w - y_v}{x_w - x_v} x_v$.

Si on multiplie la dernière expression par $(x_w - x_v)$ on obtient $a = (y_w - y_v)$, $b = (x_v - x_w)$ et $c = (x_w - x_v)y_v - (y_w - y_v)x_v$ qui est valable aussi si $x_v = x_w$.

Correction

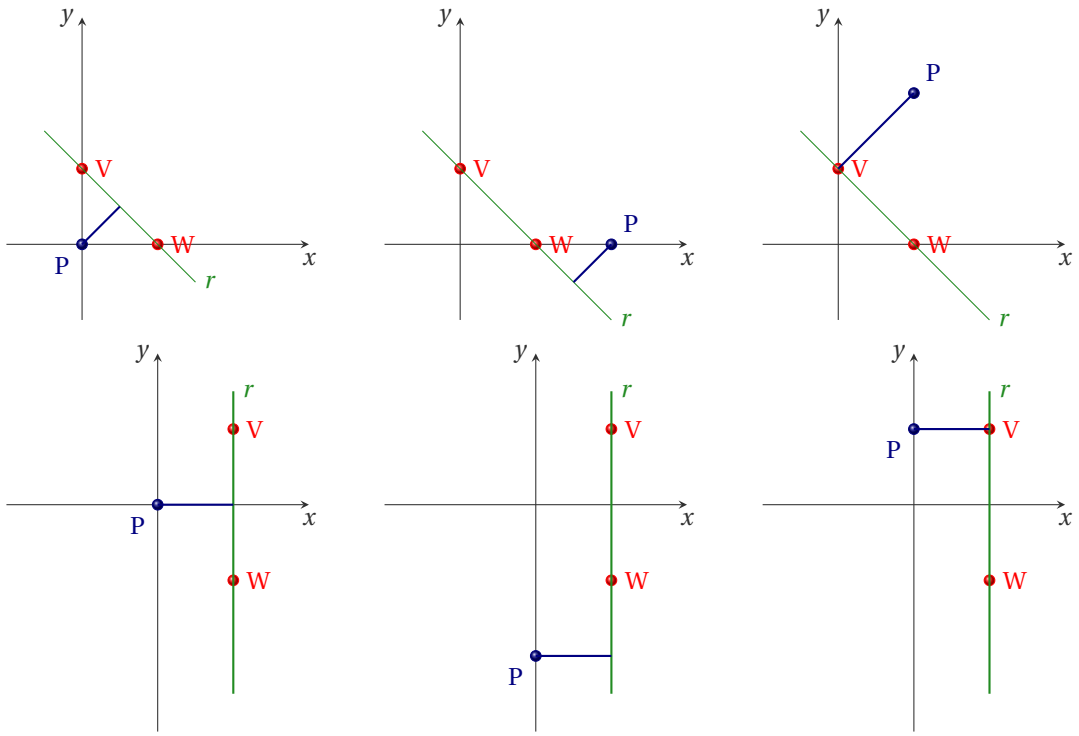
Tout d'abord, pour calculer l'équation de la droite passant par les points $\mathrm{V} = (x_v, y_v)$ et $\mathrm{W} = (x_w, y_w)$, on peut simplement écrire

$$\begin{cases} ax_v + by_v + c = 0, \\ ax_w + by_w + c = 0, \end{cases} \quad \text{ce qui équivaut à} \quad \begin{cases} a(x_v - x_w) + b(y_v - y_w) = 0, \\ ax_w + by_w + c = 0, \end{cases}$$

Un choix possible pour a et b est $a = -(y_v - y_w)$ et $b = (x_v - x_w)$ ainsi $c = -ax_v - by_v = (x_v y_w - y_v x_w)$. En utilisant ces coefficients, la fonction `dist_point_droite` peut être définie comme suit:

```
from math import sqrt

def dist_point_droite(V,W,P):
    a = V[1]-W[1]
    b = W[0]-V[0]
    c = V[0]*W[1]-V[1]*W[0]
    return abs(a*P[0]+b*P[1]+c)/sqrt(a**2+b**2)

TESTS = []
TESTS.append( [(0,1), (1,0), (0,0)] )
TESTS.append( [(0,1), (1,0), (2,0)] )
TESTS.append( [(0,1), (1,0), (1,2)] )
TESTS.append( [(1,1), (1,-1), (0, 0)] )
TESTS.append( [(1,1), (1,-1), (0,-2)] )
TESTS.append( [(1,1), (1,-1), (0, 1)] )
TESTS.append( [(-1,1), (1,1), ( 0,0)] )
TESTS.append( [(-1,1), (1,1), (-3,0)] )
TESTS.append( [(-1,1), (1,1), ( 3,0)] )
for V, W, P in TESTS :
    print(f"{V=}, {W=}, {P=}, d_r={dist_point_droite(V,W,P)}")
```

```
V=(0, 1), W=(1, 0), P=(0, 0), d_r=0.7071067811865475
V=(0, 1), W=(1, 0), P=(2, 0), d_r=0.7071067811865475
V=(0, 1), W=(1, 0), P=(1, 2), d_r=1.414213562373095
V=(1, 1), W=(1, -1), P=(0, 0), d_r=1.0
V=(1, 1), W=(1, -1), P=(0, -2), d_r=1.0
V=(1, 1), W=(1, -1), P=(0, 1), d_r=1.0
V=(-1, 1), W=(1, 1), P=(0, 0), d_r=1.0
V=(-1, 1), W=(1, 1), P=(-3, 0), d_r=1.0
V=(-1, 1), W=(1, 1), P=(3, 0), d_r=1.0
```



Exercice Bonus 7.23 (Module `math` – Distance point segment)
On se donne trois points V, W et P dans un plan cartésien. On va complexifier l'exercice 7.22 : écrire une fonction `dist_point_segment` qui a les même paramètres en entrée et renvoi la distance de P du segment d'extrémités V et W.

Correction
La droite r passant par les points $\mathrm{V} = (x_v, y_v)$ et $\mathrm{W} = (x_w, y_w)$ a pour equation

$$\underbrace{(y_w - y_v)}_{a} x + \underbrace{(x_v - x_w)}_{b} y + \underbrace{(x_v y_w - y_v x_w)}_{c} = 0.$$

Pour calculer la distance d_s du point P au segment, il faut d'abord calculer le point M projection de P sur r, c'est à dire l'intersection de la droite r avec la droite perpendiculaire qui passe par P. Cette droite a pour équation

$$\underbrace{(x_w - x_v)}_{a_1} x + \underbrace{(y_w - y_v)}_{b_1} y + \underbrace{-\big(y_p(y_w - y_v) - x_p(x_w - x_v)\big)}_{c_1} = 0.$$

Donc le point M a pour coordonnées :

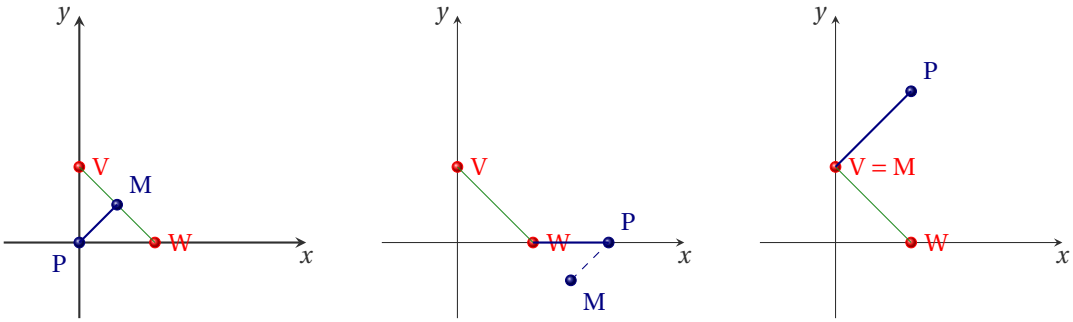
$$\begin{cases} ax + by + c = 0, \\ a_1 x + b_1 y + c_1 = 0, \end{cases} \quad \rightsquigarrow \quad (x, y) = \left(\frac{bc_1 - b_1 c}{ab_1 - a_1 b}, \frac{ca_1 - c_1 a}{ab_1 - a_1 b} \right).$$

Si M appartient au segment, alors $d_s = d_r$ (distance de P de la droite r comme à l'exercice 7.22) ; sinon $d_s = \min\{d(\mathrm{P},\mathrm{W}), d(\mathrm{P},\mathrm{V})\}$.

```python
from math import dist

def dist_point_segment(V,W,P):
    a = V[1]-W[1]
    b = W[0]-V[0]
    c = V[0]*W[1]-V[1]*W[0]
    # droite perpendiculaire
    a1 = W[0]-V[0]
    b1 = W[1]-V[1]
    c1 = -P[1]*(W[1]-V[1])-P[0]*(W[0]-V[0])
    # projection de P = intersection des deux droites
    M = (b*c1 - b1*c)/(a*b1 - a1*b), (-a*c1 + a1*c)/(a*b1 - a1*b)
    # M dans le segment ?
    is_bet_x = V[0]<=M[0]<=W[0] or W[0]<=M[0]<=V[0]
    is_bet_y = V[1]<=M[1]<=W[1] or W[1]<=M[1]<=V[1]
    if is_bet_x and is_bet_y :
        return  dist(P,M)
    else :
        return min( [dist(P,V),dist(P,W)] )

TESTS = []
TESTS.append( [(0,1), (1,0), (0,0)] )
TESTS.append( [(0,1), (1,0), (2,0)] )
TESTS.append( [(0,1), (1,0), (1,2)] )
TESTS.append( [(1,1), (1,-1), (0, 0)] )
TESTS.append( [(1,1), (1,-1), (0,-2)] )
TESTS.append( [(1,1), (1,-1), (0, 1)] )
TESTS.append( [(-1,1), (1,1), ( 0,0)] )
TESTS.append( [(-1,1), (1,1), (-3,0)] )
TESTS.append( [(-1,1), (1,1), ( 3,0)] )
for V, W, P in TESTS :
    print(f"{V=}, {W=}, {P=}, d_s={dist_point_segment(V,W,P)}")
```

```
V=(0, 1), W=(1, 0), P=(0, 0), d_s=0.7071067811865476
V=(0, 1), W=(1, 0), P=(2, 0), d_s=1.0
V=(0, 1), W=(1, 0), P=(1, 2), d_s=1.4142135623730951
V=(1, 1), W=(1, -1), P=(0, 0), d_s=1.0
V=(1, 1), W=(1, -1), P=(0, -2), d_s=1.4142135623730951
```

```
V=(1, 1), W=(1, -1), P=(0, 1), d_s=1.0
V=(-1, 1), W=(1, 1), P=(0, 0), d_s=1.0
V=(-1, 1), W=(1, 1), P=(-3, 0), d_s=2.23606797749979
V=(-1, 1), W=(1, 1), P=(3, 0), d_s=2.23606797749979
```



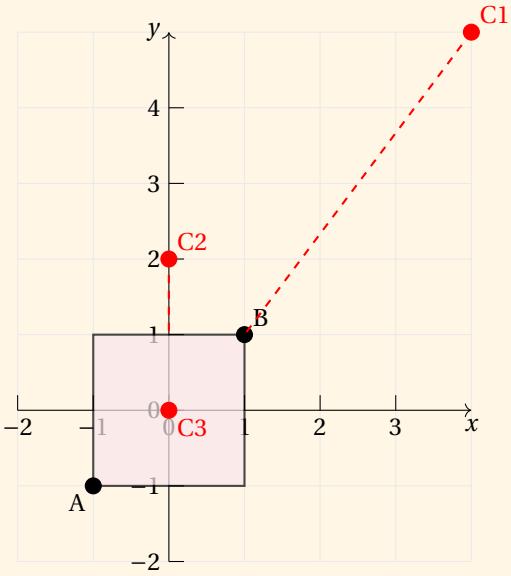
Exercice Bonus 7.24 (Module `math` – Distance point rectangle)

On se donne deux points $A = (x_A, y_A)$ et $B = (x_B, y_B)$ situés aux coins opposés d'un rectangle. Trouver la distance euclidienne minimale entre un autre point $C = (x_C, y_C)$ et ce rectangle. On supposera toujours $x_A < x_B$ et $y_A < y_B$.

Exemple : soit un carré centré à l'origine défini par $A = (-1, -1)$ et $B = (1, 1)$.

Si $C = (4, 5)$ alors la distance est 3 (le point du carré le plus proche de C est B).

Si $C = (0, 2)$ alors la distance est 1 (le point du carré le plus proche de C est $(0, 1)$).



Correction

On utilise la fonction de l'exercice 7.23 qui calcule la distance entre un point et un segment (ici les 4 coté du rectangle) :

```python
from math import dist

def dist_point_segment(V,W,P):
    a = V[1]-W[1]
    b = W[0]-V[0]
    c = V[0]*W[1]-V[1]*W[0]
    # droite perpendiculaire
    a1 = W[0]-V[0]
    b1 = W[1]-V[1]
    c1 = -P[1]*(W[1]-V[1])-P[0]*(W[0]-V[0])
    # projection de P = intersection des deux droites
    M = (b*c1 - b1*c)/(a*b1 - a1*b), (-a*c1 + a1*c)/(a*b1 - a1*b)
    # M dans le segment ?
    is_bet_x = V[0]<=M[0]<=W[0] or W[0]<=M[0]<=V[0]
```

```
    is_bet_y = V[1]<=M[1]<=W[1] or W[1]<=M[1]<=V[1]
    if is_bet_x and is_bet_y :
        return  dist(P,M)
    else :
        return min( [dist(P,V),dist(P,W)] )

def distance_point_rectangle(A, B, C):
    if A[0] <= C[0] <= B[0] and A[1] <= C[1] <= B[1]:
        return 0  # Le point est à l'intérieur du rectangle, distance = 0

    coins = [(A[0], A[1]), (B[0], A[1]), (A[0], B[1]), (B[0], B[1])]
    distances = [ dist_point_segment(coins[0],coins[1],C), dist_point_segment(coins[1],coins[2],C),
    ↪  dist_point_segment(coins[2],coins[3],C), dist_point_segment(coins[3],coins[0],C) ]

    return min(distances)

# Exemples
A, B = [-1, -1] , [1, 1]
C1 = [4, 5]
C2 = [0, 2]
C3 = [0, 0]

print(f"La distance entre C1 et le rectangle est {distance_point_rectangle(A, B, C1)}")
print(f"La distance entre C2 et le rectangle est {distance_point_rectangle(A, B, C2)}")
print(f"La distance entre C3 et le rectangle est {distance_point_rectangle(A, B, C3)}")
```

```
La distance entre C1 et le rectangle est 5.0
La distance entre C2 et le rectangle est 1.0
La distance entre C3 et le rectangle est 0
```

Exercice Bonus 7.25 (Module `math` – Longueur d'une portion de courbe)

On s'intéresse ici à l'approximation de la longueur d'une portion de courbe. Considérons le graphe d'une fonction continue $f\colon [a,b] \to \mathbb{R}$. On peut approcher la courbe par une succession de n segments reliant des points régulièrement espacés en abscisse. On pose

$$h = \tfrac{b-a}{n}, \qquad x_k = a + kh \quad (0 \le k \le n).$$

Les points de la courbe sont alors

$$(x_0, f(x_0)),\ (x_1, f(x_1)), \ldots,\ (x_n, f(x_n)).$$

La longueur du segment reliant $(x_k, f(x_k))$ à $(x_{k+1}, f(x_{k+1}))$ est donnée par le théorème de Pythagore :

$$d_k = \sqrt{(x_{k+1} - x_k)^2 + \big(f(x_{k+1}) - f(x_k)\big)^2}.$$

La longueur de la courbe peut alors être approchée par la somme des longueurs des segments :

$$\ell_{[a,b]}(f) \approx \sum_{k=0}^{n-1} d_k.$$

Écrire une fonction `longueur_courbe` qui prend en entrée $n \ge 1$, a, b et f et renvoi la longueur $\ell_{[a,b]}(f)$.

Exemple : si $f(x) = \sqrt{1-x^2}$ sur $[-1,1]$, il s'agit d'un demi cercle de rayon 1, dont la longueur exacte est $\ell_{[-1,1]}(f) = \pi$. Vous vérifierez que pour $n = 1$, la fonction renvoie 2 (un seul segment) et que lorsque $n \to \infty$, la valeur retournée tend vers π.

Correction

```
from math import dist, sqrt, pi

def longueur_courbe(n, a, b, f):
    h  = (b - a) / n
    xx = [ a + k*h for k in range(n+1) ]
    yy = [ f(x) for x in xx ]
```

```
    dd = [ dist( [xx[k],yy[k]] , [xx[k+1],yy[k+1]] ) for k in range(n) ]
    return sum(dd)

# TESTS
f = lambda x : sqrt(1 - x**2)
for n in [2, 4, 10, 50, 200, 1000]:
    print(f"{n = }, longueur = {longueur_courbe(n, -1, 1, f)}")

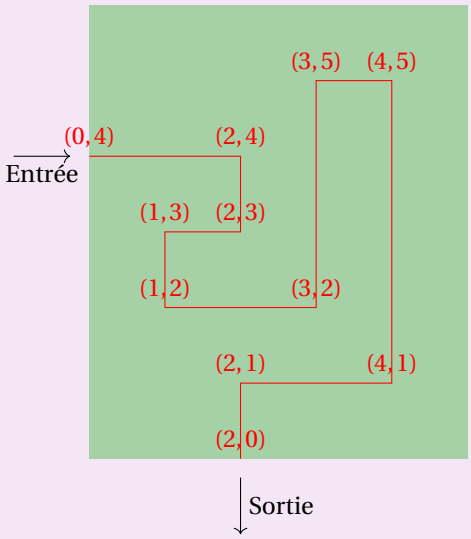
print(f"Longueur exacte = {pi}")
```

```
n = 2, longueur = 2.8284271247461903
n = 4, longueur = 3.035276180410083
n = 10, longueur = 3.115105950558347
n = 50, longueur = 3.139237702364473
n = 200, longueur = 3.1412985671602187
n = 1000, longueur = 3.1415663562164795
Longueur exacte = 3.141592653589793
```

Cette approximation devient d'autant meilleure que n est grand : l'algorithme reproduit exactement la démarche géométrique étudiée dans l'énoncé.

Exercice Bonus 7.26 (Pydéfis – La biche de Cyrénée)

Histoire : pour son troisième travail, Eurysthée demanda à Hercule de capturer la biche aux pieds d'airain qui s'était enfuie de l'attelage d'Artémis. La difficulté pour Hercule était de capturer la biche sans la blesser, sous peine d'essuyer la colère d'Artémis.



Il décida donc de l'épuiser en la poursuivant dans les bois d'Oénoée. Son plan était clair : il commença par aménager les carrefours afin que la biche, à chaque embranchement, n'ait qu'un seul choix de parcours. Il construisit un plan qui ressemblait un peu à celui-ci, encore que le vrai plan était beaucoup plus grand :

Les chemins étaient tous parfaitement orthogonaux, direction Nord-Sud ou Est-Ouest. Puis il nota les positions des carrefours, dans l'ordre où la biche devait les parcourir. Dans l'exemple qui précède, il aurait noté les coordonnées de tous les points ainsi : $(0,4)$, $(2,4)$, $(2,3)$, $(1,3)$, $(1,2)$, $(3,2)$, $(3,5)$, $(4,5)$, $(4,1)$, $(2,1)$, $(2,0)$. Enfin, il dût choisir ses chaussures. Hercule étant un athlète très méthodique, sa paire de chaussures devait être choisie en fonction de la distance précise qu'il aurait à parcourir.

Problème : dans l'exemple qui précède, en suivant le parcours indiqué par les positions des carrefours, notées en kilomètres, Hercule aurait parcouru 18 kilomètres, ce que l'on peut voir sur le dessin, mais aussi calculer à partir du relevé de coordonnées. Le bois d'Oénoée était en réalité beaucoup plus grand que ce qui est indiqué précédemment. L'entrée du problème est le relevé des positions des carrefours, tous les nombres ayant été mis à la suite, sans les parenthèses, et les valeurs étant données en kilomètres. Aide Hercule en calculant pour lui la distance qu'il aura à parcourir dans le bois, à la poursuite de la biche.

Source : https://pydefis.callicode.fr/defis/Herculito03Biche/txt

Exercice 7.27 (Temps de parcours d'une liste)

Considérez une liste L de 10^7 éléments. Mesurez le temps nécessaire pour parcourir cette liste de deux manières : d'abord en accédant à chaque élément par son indice, puis en accédant directement à chaque élément dans la boucle. Quelle méthode est la plus rapide ?

Correction

```
from time import perf_counter
```

```
N = 10**7  # Taille de la liste
L = list(range(N))

# Mesure du temps de parcours par indice
debut = perf_counter()
for i in range(N):
    val = L[i]
fin = perf_counter()
print(f"Parcours par indice : temps d'exécution = {fin-debut:g}s")

# Mesure du temps de parcours par élément
debut = perf_counter()
for val in L:  # Accès direct aux éléments
    pass
fin = perf_counter()
print(f"Parcours par élément : temps d'exécution = {fin-debut:g}s")
```

```
Parcours par indice : temps d'exécution = 2.54035s
Parcours par élément : temps d'exécution = 0.635786s
```

Cette méthode mesure le temps d'exécution en utilisant un chronomètre haute résolution. Elle est simple et directe, mais elle ne fournit qu'**une seule mesure du temps d'exécution**, ce qui peut être influencé par des variations aléatoires.

Le module `timeit` fournit une méthode qui exécute le code plusieurs fois et calcule un temps moyen. Cela permet d'obtenir des résultats plus fiables et précis en réduisant l'impact des variations aléatoires :

```
import timeit

# Codes à mesurer
code_to_test_1 = """
for i in range(N):
    val = L[i]
"""

code_to_test_2 = """
for val in L:
    pass
"""

# Paramètres communs
N = 10**7
L = range(N)
config = f"{N=}; {L=}"
repetitions = 10

# Mesurer les temps d'exécution
execution_time_1 = timeit.timeit(code_to_test_1, setup=config, number=repetitions)
print(f"Parcours par indice : temps d'exécution moyen = {execution_time_1 / repetitions} secondes")

execution_time_2 = timeit.timeit(code_to_test_2, setup=config, number=repetitions)
print(f"Parcours par élément : temps d'exécution moyen = {execution_time_2 / repetitions} secondes")
```

```
Parcours par indice : temps d'exécution moyen = 1.4670125504009774 secondes
Parcours par élément : temps d'exécution moyen = 0.2699601950997021 secondes
```

Exercice 7.28 (Réduire une fraction : module `math` puis modules `fractions` et `sympy`)

Compléter la fonction `reduce_fraction(numerator, denominator)` qui prend comme paramètres deux entiers strictement positifs représentant le numérateur et le dénominateur d'une fraction et qui renvoie le numérateur et le dénominateur de la fraction réduite. Par exemple, si les paramètres transmis à la fonction sont 6 et 63, la fonction doit renvoyer 2 et 21.

On pourra utiliser la fonction gcd du module `math`.

Bonus : comparer votre résultat à celui de la fonction `Fraction` du module `fractions` ou de la fonction `Rational` du module `sympy`.

Correction

```
# Avec le module math
from math import gcd
def reduce_fraction(numerator, denominator):
    # trouver le plus grand diviseur commun des deux nombres
    NDgcd = gcd(numerator,denominator)
    # diviser le numérateur et le dénominateur par le plus grand diviseur commun
    numerator //= NDgcd
    denominator //= NDgcd
    return (numerator, denominator)

# Avec le module fractions
from fractions import Fraction
def reduce_fraction_bis(numerator, denominator):
    return Fraction(numerator, denominator)

# Avec le module sympy
from sympy import Rational
def reduce_fraction_ter(numerator, denominator):
    return Rational(numerator, denominator)

from random import *
TESTS = [ (6,63) , (1,2) , (25,5),  (randrange(1,100),randrange(1,100)) ]
for N,D in TESTS:
    reduced_numerator,reduced_denominator = reduce_fraction(N,D)
    print(f'{N}/{D} = {reduced_numerator}/{reduced_denominator} (avec le module math), =
    ↪ {reduce_fraction_bis(N,D)} avec le module fractions et = {reduce_fraction_ter(N,D)} avec le
    ↪ module sympy')
```

```
6/63 = 2/21 (avec le module math), = 2/21 avec le module fractions et = 2/21 avec le module sympy
1/2 = 1/2 (avec le module math), = 1/2 avec le module fractions et = 1/2 avec le module sympy
25/5 = 5/1 (avec le module math), = 5 avec le module fractions et = 5 avec le module sympy
97/77 = 97/77 (avec le module math), = 97/77 avec le module fractions et = 97/77 avec le module sympy
```

Exercice Bonus 7.29 (Module `Fraction` – Triangles semblables)

Considérons deux listes contenant chacune trois tuples représentant les sommets d'un triangle. Écrire une fonction qui renvoie `True` si les deux triangles sont semblables, `False` sinon.

Source : https://py.checkio.org/en/mission/similar-triangles/

Correction

Tester `x == flottant` est presque toujours une erreur, on utilisera plutôt `abs(x-flottant)<1.e-10` par exemple.

Si on suppose que les coordonnées sont des valeurs dans $\mathbb{Q}$, on peut comparer le carré des distances (en évitant les racines carrées) et utiliser le module `Fraction` pour comparer les rapports, comme dans l'exemple ci-dessous :

```
from fractions import Fraction

def similar_triangles(T,V):
    dist2 = lambda p1,p2 : (p1[0]-p2[0])**2+(p1[1]-p2[1])**2
    a1,b1,c1 = sorted( [dist2(T[0],T[1]) , dist2(T[1],T[2]) , dist2(T[2],T[0])]  )
    a2,b2,c2 = sorted( [dist2(V[0],V[1]) , dist2(V[1],V[2]) , dist2(V[2],V[0])]  )
    return Fraction(a2,a1)==Fraction(b2,b1)==Fraction(c2,c1)
    # sans le module Fraction on écrira
    # toll = 1.e-12
    # return abs(a2*b1-a1*b2)<toll and abs(b2*c1-b1*c2)<toll and abs(a2*c1-a1*c2)<toll

# TESTS
TEST = []
TEST.append( ( [(0, 0), (1, 2), (2, 0)] , [(3, 0), (4, 2), (5, 0)] ) )
TEST.append( ( [(0, 0), (1, 2), (2, 0)] , [(3, 0), (4, 3), (5, 0)] ) )
TEST.append( ( [(0, 0), (1, 2), (2, 0)] , [(2, 0), (4, 4), (6, 0)] ) )
TEST.append( ( [(0, 0), (0, 3), (2, 0)] , [(3, 0), (5, 3), (5, 0)] ) )
TEST.append( ( [(1, 0), (1, 2), (2, 0)] , [(3, 0), (5, 4), (5, 0)] ) )
```

```
TEST.append( ( [(1, 0), (1, 3), (2, 0)] , [(3, 0), (5, 5), (5, 0)] ) )

print(*[similar_triangles(T,V) for T,V in TEST],sep=", ")

True, False, True, True, True, False
```

Exercice 7.30 (Module `random` - Jeu de dé)

On lance un dé. Si le numéro est 1, 5 ou 6, alors c'est gagné, sinon c'est perdu.

1. Écrire un script simulant ce jeu 8 fois et qui affiche "Gagné!" ou "Perdu".
2. Simuler ensuite le jeu 10 000 fois et afficher combien de fois on a gagné.

Correction

On simule 10 tentatives :

```
from random import randint
T = 8
Tirages = [randint(1,6) for t in range(T)]
for t in Tirages:
    if t in [1,5,6] :
        print('Gagné!')
    else:
        print('Perdu')
```

```
Perdu
Gagné!
Gagné!
Gagné!
Perdu
Gagné!
Perdu
Perdu
```

On simule 10 000 tentatives et on affiche seulement combien de fois on a gagné :

```
from random import randint
T = 10000
Tirages = [randint(1,6) for t in range(T)]

# Méthode 1
#Gagne=0
#for t in Tirages:
#    if t in [1,5,6] :
#        Gagne+=1

# Méthode 2
# Gagne = sum([1 for t in Tirages if t in [1,5,6]])

# Méthode 3
Gagne = sum([t in [1,5,6] for t in Tirages])

print(f"Sur {T} tirages on a gagné {Gagne} fois.")

Sur 10000 tirages on a gagné 4971 fois.
```

Version *one-liner* :

```
from random import randint
T = 10000
Gagne=sum( [ randint(1,6) in [1,5,6]   for t in range(T) ] )
print(f"Sur {T} tirages on a gagné {Gagne} fois.")

Sur 10000 tirages on a gagné 5010 fois.
```

Exercice 7.31 (Module `random` - Tirage de nombres aléatoires)

Écrivez une boucle qui tire des nombres aléatoires dans l'intervalle $[0,1]$ jusqu'à ce que la somme des nombres tirés soit supérieure à 1 et comptez le nombre de tirages nécessaires. Vérifiez que le nombre moyen de tirages nécessaire est proche de la constante mathématique e.

Correction

```
from random import uniform

def tirage_somme_sup_1():
    somme = 0
    nb_tirages = 0
    while somme <= 1:
        somme += uniform(0, 1)
        nb_tirages += 1
    return nb_tirages

# Tests
TESTS = [ tirage_somme_sup_1() for _ in range(1000) ]
print(f"Moyenne = {sum(TESTS)/len(TESTS)}")
```

```
Moyenne = 2.677
```

Exercice 7.32 (Module `random` - Calcul de fréquences)

On tire 10000 nombres au hasard dans l'intervalle $[0,1]$. À chaque tirage, on se demande si l'événement A : «le nombre tiré au hasard appartient à l'intervalle $[1/7, 5/6]$» est réalisé.

1. Combien vaut la probabilité p de A ?
2. Calculer la fréquence de réalisation de A.

Correction

L'énoncé ne précise pas selon quel hasard les nombres sont tirés dans l'intervalle $[0,1]$. En fait, il est sous-entendu qu'ils sont tirés selon la loi uniforme sur $[0,1]$. Dans ces conditions, par définition de cette loi, la probabilité de l'événement A est la longueur de $[1/7, 5/6]$, soit $p = 5/6 - 1/7 = 29/42$.

```
from random import random
T = 10000
Tirages = [random() for _ in range(T)] # _ car variable muette
OuiA = [1 for t in Tirages if 1/7<t<5/6]
print(f"Fréquence théorique = {29/42}, Fréquence moyenne = {sum(OuiA)/T}")
```

```
Fréquence théorique = 0.6904761904761905, Fréquence moyenne = 0.6824
```

Exercice 7.33 (Module `random` - Puce)

Une puce fait des bonds successifs indépendants les uns des autres en ligne droite. La longueur de chaque bond est aléatoire. On considère que c'est un nombre choisi au hasard dans l'intervalle $[0,5[$, l'unité de longueur étant le cm. On note la distance totale parcourue au bout de m bonds. On répète cette expérience n fois. Calculer la distance moyenne parcourue au cours de ces n expériences.

Correction

Avec le script

```
from random import  randint
n = 1000 # nombre de tirage
m = 20 # nombre de bond par tirage
L = [ sum([randint(0,5) for i in range(m)]) for j in range(n)]
print(sum(L)/n)
```

on trouve

```
50.512
```

Explications :

- `[randint(0,5) for i in range(m)]` est une liste qui contient les m bonds d'une expérience. En faisant la somme, on obtient la distance parcourue lors de cette expérience.
- `L[j]` contient donc la distance parcourue à la j-ème expérience.
- On voit que la distance moyenne parcourue tend (*i.e.* si on augment n) vers $\frac{(5-0)}{2}m$.

Exercice Bonus 7.34 (Module `random` - Le nombre mystère)

Écrire un script où l'ordinateur choisit un nombre mystère entre 0 et 100 (inclus) au hasard et le joueur doit deviner ce nombre en suivant les indications «plus grand» ou «plus petit» données par l'ordinateur. Le joueur a sept tentatives pour trouver la bonne réponse. Programmer ce jeu.

Pour que l'ordinateur choisisse un entier aléatoirement dans l'intervalle $[0;100]$ on écrira

```
import random
N = random.randint(0,100)
```

Pour affecter à la variable `j` la valeur que le joueur tape au clavier on écrira

```
j=int(input('Quel nombre proposes-tu ?'))
```

Correction

```
import random
N = random.randint(0,100)

j = -1
i = 0
while j!=N and i<7:
    i += 1
    j = int(input('Quel nombre proposes-tu ? '))
    if j>N:
        print('Le nombre à trouver est plus petit')
    elif j<N:
        print('Le nombre à trouver est plus grand')
    else:
        print('Bravo')
        break
    if i==7:
        print('Dommage')
        break
print(f"N = {N}, J = {j}, Tentatives = {i}")
```

La techniques optimale pour jouer contre l'ordinateur est d'utiliser une méthode dichotomique :

- on pose $a = 0$, $b = 100$ et on propose $c = \mathrm{E}\left(\frac{a+b}{2}\right) = 50$,
- si le nombre mystère est plus petit que c on posera $b = c$, sinon $a = c$ et on recommence avec $c = \mathrm{E}\left(\frac{a+b}{2}\right)$.

Cette stratégie se termine lorsque $b = a$. Comme à chaque étape l'intervalle de recherche est divisé en deux parties, si initialement on a $b - a = d$, après n étapes on aura $b - a = d \times 2^{-n}$ ainsi $d \times 2^{-n} \leq 1$ pour $n \geq \log_2(d)$. Ici $d = 100$ et $\log_2(d) = 6.643856189774725$: en 7 étapes (n de 0 à 6) on trouvera forcement le nombre mystère.

Exercice Bonus 7.35 (Module `random` - Yahtzee)

Écrire un script où l'ordinateur simule le lancé de trois dés en même temps et il continue tant que il obtient un "yahtzee" (les trois dés obtiennent tous les trois 6) ou si le nombre de lancés sans obtenir de yahtzee est supérieur à 100.

Correction

```
from random import randint
d1, d2, d3 = 0, 0, 0
count = 1
while (d1+d2+d3 < 18 and count <= 100):
    d1 = randint(1,6)
    d2 = randint(1,6)
    d3 = randint(1,6)
    count += 1
    # print(f"{d1}, {d2}, {d3}")

if d1+d2+d3 == 18:
    print (f"Yahtzee avec {count-1} tentatives")
else:
    print( "no Yahtzee :( ")
```

Bonus : on peut imprimer les dés :

```
print([chr(0x2680+i) for i in range(6)])
```

Exercice 7.36 (Module `math` – Cylindre)

Fabriquer une fonction qui calcule le volume d'un cylindre de révolution de hauteur h et dont la base est un disque de rayon r.

Correction

On écrit la fonction soit avec `def` soit avec une `lambda` function, puis on valide la fonction avec un test dont on connaît le résultat :

```
>>> import math
>>> volume = lambda h,r : (r**2 * h * math.pi)
>>> h,r = 1,1
>>> print(f'h = {h} cm, r = {r} cm, v = {volume(h,r):.5f} cm2')
h = 1 cm, r = 1 cm, v = 3.14159 cm2
```

Exercice 7.37 (Module `math` – Formule d'Héron)

Pour le calcul de l'aire $\mathscr{A}(\mathrm{T})$ d'un triangle T de coté a, b et c, on peut utiliser la formule d'Héron :

$$\mathscr{A}(\mathrm{T}) = \sqrt{p(p-a)(p-b)(p-c)}$$

où p est le demi-périmètre de T. Écrire une fonction qui implémente cette formule. La tester en la comparant à la solution exacte (calculée à la main).

Vérifier ensuite l'inégalité de Weitzenböck :

$$\mathscr{A}(\mathrm{T}) \leq \frac{a^2+b^2+c^2}{4\sqrt{3}}$$

Correction

On choisi d'utiliser la fonction `sqrt()` du module `math`. Il est cependant possible d'utiliser `**0.5` sans utiliser d'autres modules.

```
import math
def Heron(a,b,c):
    p = (a+b+c)/2
    return math.sqrt(p*(p-a)*(p-b)*(p-c))

# TEST-1
(a,b,c) = (5,4,3)
He = Heron(a,b,c)
Ex = b*c/2
print(f"Erone = {He:.5f}, Exacte = {Ex:.5f}; Weitzenböck = {(a**2+b**2+c**2)/(4*math.sqrt(3)):.5f}")

# TESTS
# on choisit des cas où le calcul de la solution exacte est simple
for a,b,c in [(5,2.5,2.5),(5,3,3),(5,5,5)]:
    He = Heron(a,b,c)
    Ex = a/2*math.sqrt(b**2-(a/2)**2)
    Err = He-Ex
    print(f"Erone = {He:.5f}, Exacte = {Ex:.5f}, Erreur = {Err:.5f}; Weitzenböck =
    ↪ {(a**2+b**2+c**2)/(4*math.sqrt(3)):.5f}")
```

```
Erone = 6.00000, Exacte = 6.00000; Weitzenböck = 7.21688
Erone = 0.00000, Exacte = 0.00000, Erreur = 0.00000; Weitzenböck = 5.41266
Erone = 4.14578, Exacte = 4.14578, Erreur = 0.00000; Weitzenböck = 6.20652
Erone = 10.82532, Exacte = 10.82532, Erreur = -0.00000; Weitzenböck = 10.82532
```

Exercice 7.38 (Module `math` – Formule de Kahan)

Pour le calcul de l'aire $\mathscr{A}(\mathrm{T})$ d'un triangle T de coté a, b et c avec $a \geq b \geq c$, William Kahan a proposé une formule plus stable que celle d'Héron :

$$\mathrm{S} = \frac{1}{4}\sqrt{[a+(b+c)][c-(a-b)][c+(a-b)][a+(b-c)]}.$$

Écrire une fonction qui implémente cette formule. La tester en la comparant à la solution exacte (calculée à la main).

Correction

```
import math
def Khan(a,b,c):
    c,b,a = sorted([a,b,c])
    return 0.25*math.sqrt( (a+(b+c))*(c-(a-b))*(c+(a-b))*(a+(b-c))  )

# TEST-1
(a,b,c) = (5,4,3)
He = Khan(a,b,c)
Ex = b*c/2
print(f"Khan={He:.5f}, Exacte={Ex:.5f}")

# TESTS
# on choisit des cas où le calcul de la solution exacte est simple
for a,b,c in [(2.5,5,2.5),(5,3,3),(5,5,5)]:
    He = Khan(a,b,c)
    Ex = a/2*math.sqrt(b**2-(a/2)**2)
    Err = He-Ex
    print(f"Khan = {He:.5f}, Exacte = {Ex:.5f}, Erreur = {Err:.5f}")
```

```
Khan=6.00000, Exacte=6.00000
Khan = 0.00000, Exacte = 6.05154, Erreur = -6.05154
Khan = 4.14578, Exacte = 4.14578, Erreur = 0.00000
Khan = 10.82532, Exacte = 10.82532, Erreur = -0.00000
```

Exercice 7.39 (Module `math` – Cercle circonscrit)

Écrire (et tester) une fonction qui prend en entrée la longueur des trois cotés d'un triangle et renvoie le rayon du cercle circonscrit.

Rappel : si on note a, b et c les longueurs des trois cotés du triangle et $\mathscr{A}$ son aire, alors le rayon du cercle circonscrit est donné par

$$r = \frac{abc}{4\mathscr{A}}.$$

Correction

On va écrire d'abord une fonction qui calcule l'aire du triangle à partir des longueurs des trois cotés (par exemple avec la formule d'Héron, *cf.* exercice 7.37), puis écrire une fonction qui calcule le rayon.

```
import math

def Heron(a,b,c):
    p = (a+b+c)/2
    return math.sqrt(p*(p-a)*(p-b)*(p-c))
    
def circumscribed(a,b,c):
    A = Heron(a,b,c)
    return a*b*c/(4*A)

# TEST-1
(a,b,c) = (1,1,1)
r = circumscribed(a,b,c)
Ex = 1/math.sqrt(3)
print(f"r = {r:.5f}, Exacte = {Ex:.5f}")
```

```
# TEST-2
(a,b,c) = (3,4,5)
r = circumscribed(a,b,c)
Ex = 2.5
print(f"r = {r:.5f}, Exacte = {Ex:.5f}")

# TEST-3
(a,b,c) = (1,1,math.sqrt(3))
r = circumscribed(a,b,c)
Ex = 1
print(f"r = {r:.5f}, Exacte = {Ex:.5f}")


# TEST-4
(a,b,c) = (2,2,math.sqrt(7))
r = circumscribed(a,b,c)
Ex = 4/3
print(f"r = {r:.5f}, Exacte = {Ex:.5f}")
```

```
r = 0.57735, Exacte = 0.57735
r = 2.50000, Exacte = 2.50000
r = 1.00000, Exacte = 1.00000
r = 1.33333, Exacte = 1.33333
```

Exercice Bonus 7.40 (Centre et rayon d'un cercle pour un arc donné)

L'objectif de cet exercice est d'écrire une fonction `find_center(p1, p2, angle)` qui renvoie le centre et le rayon d'un cercle, à partir de deux points sur un arc et l'angle entre eux.

Entrées : `p1` et `p2` sont deux liste contenantes les coordonnées $[x, y]$ des deux points sur l'arc du cercle, `angle` est l'angle entre les deux points (en degrés).

Sortie : `xc` et `yc` sont les coordonnées du centre du cercle, `r` son rayon.

Correction

```
import math

def find_center(p0, p1, angle):

    x0, y0 = p0
    x1, y1 = p1
    a = math.radians(angle)
    
    # Distance entre les points p1 et p2 et calcul du rayon
    d_p1p0 = math.sqrt((x1 - x0)**2 + (y1 - y0)**2)
    r = d_p1p0 / (2 * math.sin(a / 2))
    
    # Pente de la droite passant par la corde
    slope = (y1 - y0) / (x1 - x0) if x1 != x0 else float('inf')
    
    # Pente de la droite perpendiculaire à la corde
    new_slope = -1 / slope if slope != 0 else float('inf')
    
    # Point sur la droite perpendiculaire à la corde
    # passant par le centre du cercle
    xm, ym = (x1 + x0) / 2 , (y1 + y0) / 2
    
    # Distance entre (xm,ym) et le centre du cercle (xc, yc)
    d_cm = d_p1p0 / (2 * math.tan(a))

    if math.isinf(new_slope):
        xc = xm
        yc = ym - d_cm
    else:
```

```
        xc = xm - (d_cm) / math.sqrt(new_slope**2 + 1)
        yc = new_slope * (xc - xm) + ym

    return xc, yc, r

# TESTS
TESTS = [ ([1,0],[0,1],90) , ([0,1],[1,0],270) , ([1,1],[-1,1],45)  ]

for p1,p2,angle in TESTS:
    xc,yc,r = find_center(p1, p2, angle)
    print( f"{p1=}, {p2=}, {angle=} ⇝ {xc=:g}, {yc=:g}, {r=:g} " )
```

```
p1=[1, 0], p2=[0, 1], angle=90 ⇝ xc=0.5, yc=0.5, r=1
p1=[0, 1], p2=[1, 0], angle=270 ⇝ xc=0.5, yc=0.5, r=1
p1=[1, 1], p2=[-1, 1], angle=45 ⇝ xc=0, yc=-2.22045e-16, r=2.61313
```

Exercice 7.41 (Module `random` - Distance moyenne entre deux points aléatoires d'un carré)
Considérons un carré de côté 1 et plaçons en son intérieur deux points de manière aléatoire. Quelle est la distance moyenne entre deux points ?

Correction

```
from random import random
N = 1000
points = [ [random(),random(),random(),random()] for i in range(N) ]
distan = [ ((points[i][0]-points[i][1])**2+(points[i][2]-points[i][3])**2)**0.5 for i in range(N) ]
print(f"Avec {N} couples de points, en moyenne les deux points de chaque couple sont à une distance de
↪ {sum(distan)/len(distan)} l'un de l'autre.")
```

```
Avec 1000 couples de points, en moyenne les deux points de chaque couple sont à une distance de
↪ 0.5304851060858936 l'un de l'autre.
```

Depuis la version 3.8 de python, la fonction `dist()` a été ajoutée dans le module `math`. On pourra alors écrire

```
from random import random
from math import dist
N = 1000
points = [ [random(),random(),random(),random()] for i in range(N) ]
distan = [ dist( (points[i][0],points[i][1]) , (points[i][2],points[i][3]) ) for i in range(N) ]
print(f"Avec {N} couples de points, en moyenne les deux points de chaque couple sont à une distance de
↪ {sum(distan)/len(distan)} l'un de l'autre.")
```

```
Avec 1000 couples de points, en moyenne les deux points de chaque couple sont à une distance de
↪ 0.5251726933494414 l'un de l'autre.
```

Exercice 7.42 (Module `random` - Kangourou)
Un kangourou fait habituellement des bonds de longueur aléatoire comprise entre 1 et 9 mètres. [a]

- Combien de sauts devra-t-il faire pour parcourir 2000 mètres ?
- Fabriquer une fonction qui à toute distance d exprimée en mètres associe le nombre aléatoire N de sauts nécessaires pour la parcourir.
- Calculer le nombre moyen de sauts effectués par le kangourou quand il parcourt T fois la distance d.

cf. http://gradus-ad-mathematicam.fr/documents/300_Directeur.pdf

a. Il est sous-entendu que la longueur de chaque bond est la valeur prise par une variable aléatoire qui suit la loi uniforme entre 0 et 9. Il est aussi sous-entendu, comme d'habitude, que si on considère des sauts successifs, les variables aléatoires qui leur sont associées sont indépendantes.

Correction

```
from random import randint

def distanceTOpas(d):
    l = 0 # distance parcourue avant le premier saut
```

```
    b = 0 # nombre de sauts effectues avant le premier saut
    while l<=d:
        l += randint(1,9)
        b += 1
    return b,l

d = 2000
N,L = distanceTOpas(d)
print("Pour parcourir d =",d," mètres une fois il faut N =",N," bonds.")
print("La distance réelle parcourue est de ",L," mètres.")

T = 100
NN = [distanceTOpas(d)[0] for N in range(T)]
print("Pour parcourir d =",d," mètres ",T," fois il faut en moyenne N =",sum(NN)/T," bonds")
```

```
Pour parcourir d = 2000  mètres une fois il faut N = 405  bonds.
La distance réelle parcourue est de  2001  mètres.
Pour parcourir d = 2000  mètres  100  fois il faut en moyenne N = 400.74  bonds
```

Exercice 7.43 (Module `math` – sin cos)

En important seulement les fonctions nécessaires du module `math`, écrire un script qui vérifie la formule suivante pour $n = 10$:

$$\sum_{k=0}^{n} \cos(kx) = \frac{1}{2} + \frac{\sin\left(\frac{2n+1}{2}x\right)}{2\sin\left(\frac{x}{2}\right)}$$

Comparer en un point au choix.

Correction

```
>>> from math import sin, cos, pi
>>> n = 10
>>> L = lambda x : sum([cos(k*x) for k in range(n+1)])
>>> R = lambda x : 0.5*(1+sin(0.5*(2*n+1)*x)/sin(0.5*x))
>>> print(L(1),R(1))
-0.4174477464559061 -0.417447746455906
>>> # On peut évaluer la valeur absolue de la différence en plusieurs points:
>>> print([abs(L(x)-R(x)) for x in range(1,5)])
[1.1102230246251565e-16, 1.1102230246251565e-16, 1.1102230246251565e-16, 5.551115123125783e-17]
```

Exercice 7.44 (Approximations de ln)

L'algorithme de Briggs est une méthode qui permet de calculer de manière approchée le logarithme d'un nombre. Soit x un réel strictement positif. Pour approcher $\ln(x)$ on utilise la suite $(u_n)_{n\in\mathbb{N}}$ définie par

$$\begin{cases} u_0 = x, \\ u_{n+1} = \sqrt{u_n}. \end{cases}$$

On peut montrer que la suite $(w_n)_{n\in\mathbb{N}}$ définie par $w_n = 2^n(u_n - 1)$ converge vers $\ln(x)$.

Écrire une fonction qui, pour x donné, renvoie la valeur w_{N} ; N doit être le plus petit entier tel que $|u_{\mathrm{N}} - 1| > 10^{-12}$. Comparer ensuite pour différentes valeurs de x la valeur approchée obtenue par cette fonction et la valeurs `math.log(x)` du module `math`.

https://www.mathoutils.fr/grand-oral-mathematiques/grand-oral-logarithme/

Correction

```
def briggs(x,epsilon):
    n = 0
    u = x
    w = u-1
    while abs(u-1)>epsilon:
```

```
        n +=1
        u = u**0.5
        w = 2**n*(u-1)
    return w

# TESTS
from math import log
from tabulate import tabulate
T = []
T.append(["x","Briggs","math.log","erreur"])
for n in range(1,10):
    b = briggs(n,1.e-12)
    m = log(n)
    e = b-m
    T.append([n,b,m,e])

print(tabulate(T, headers="firstrow", floatfmt=".15f"))
```

```
  x             Briggs           math.log              erreur
---  -----------------  -----------------  ------------------
  1  0.000000000000000  0.000000000000000   0.000000000000000
  2  0.693115234375000  0.693147180559945  -0.000031946184945
  3  1.098388671875000  1.098612288668110  -0.000223616793110
  4  1.386230468750000  1.386294361119891  -0.000063892369891
  5  1.609375000000000  1.609437912434100  -0.000062912434100
  6  1.791503906250000  1.791759469228055  -0.000255562978055
  7  1.945800781250000  1.945910149055313  -0.000109367805313
  8  2.079101562500000  2.079441541679836  -0.000339979179836
  9  2.196777343750000  2.197224577336220  -0.000447233586220
```

Exercice 7.45 (Approximations de π)

Dans cet exercice nous allons étudier et mettre en œuvre certaines méthodes qui peuvent être utilisées pour calculer les premiers chiffres du nombre irrationnel π. Des méthodes sont basées sur des tirages aléatoires, les autres sont des algorithmes itératifs.

La valeur de π considérée comme exacte sera celle du module `math`.

Méthode 1 : Fractions

Évaluer les approximations suivantes de π et en donner la précision

$$\frac{22}{7}, \qquad \frac{355}{113}, \qquad 3+\frac{8}{60}+\frac{29}{60^2}+\frac{44}{60^3}.$$

Bonus : vérifier analytiquement que

$$\pi = \frac{22}{7} - \int_0^1 \frac{x^4(1-x)^4}{1+x^2}\,\mathrm{d}x.$$

On peut alors donner des bornes sur l'erreur commise en utilisant la valeur $\frac{22}{7}$ pour approcher π car

$$\int_0^1 \frac{x^4(1-x)^4}{2}\,\mathrm{d}x < \underbrace{\int_0^1 \frac{x^4(1-x)^4}{1+x^2}\,\mathrm{d}x}_{=\frac{22}{7}-\pi} < \int_0^1 \frac{x^4(1-x)^4}{1}\,\mathrm{d}x.$$

Méthode 2 : Racines

Évaluer les approximations suivantes de π et en donner la précision

$$\sqrt{2}+\sqrt{3}, \qquad \sqrt[3]{31}, \qquad \sqrt{63}-\sqrt{23}, \qquad \sqrt{51}-\sqrt{16}, \qquad \sqrt[15]{28658146}, \qquad \sqrt[4]{\frac{2143}{22}}.$$

Méthode 3 : Formule de Liu Hui

$$\pi \approx 768\sqrt{2-\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+1}}}}}}}}}$$

Méthode 4 : Logarithmes

$$\pi \approx \frac{\ln(640320^3+744)}{\sqrt{163}}, \qquad \pi \approx \frac{12\ln\left((2\sqrt{2}+\sqrt{10})(3+\sqrt{10})\right)}{\sqrt{190}}$$

Méthode 5 : Formule de Basel https://en.wikipedia.org/wiki/Basel_problem

$$\lim_{N\to+\infty} s(N) = \frac{\pi^2}{6}, \qquad \text{avec} \qquad s(N) \overset{\text{déf}}{=} \sum_{n=1}^{N} \frac{1}{n^2}$$

- Écrire une fonction qui, pour N donné, renvoie $s(N)$.
- Écrire un script qui calcule pour quelles valeurs de N on obtient une approximation de $\frac{\pi^2}{6}$ à 10^{-5} près.

Méthode 6 : Formule de Madhava–Leibniz

$$\pi = 4\lim_{N\to+\infty} s(N) \qquad \text{avec} \qquad s(N) \overset{\text{déf}}{=} \sum_{n=0}^{N} \frac{(-1)^n}{2n+1}.$$

- Écrire une fonction qui, pour N donné, renvoie $s(N)$.
- Écrire un script qui calcule pour quelles valeurs de N on obtient une approximation de π à 10^{-3} près. Même exercice à 10^{-4} près puis 10^{-5} près.

Méthode 7 : Formule de Madhava

$$\pi = \sqrt{12}\lim_{N\to+\infty} s(N) \qquad \text{avec} \qquad s(N) \overset{\text{déf}}{=} \sum_{n=0}^{N} \frac{1}{(-3)^n(2n+1)}.$$

- Écrire une fonction qui, pour N donné, renvoie $s(N)$.
- Écrire un script qui calcule pour quelles valeurs de N on obtient une approximation de π à 10^{-3} près. Même exercice à 10^{-4} près puis 10^{-5} près.

Méthode 8 : Formule de Bailey-Borwein-Plouffe (BBP)

$$\pi = \lim_{N\to+\infty} s(N) \qquad \text{avec} \qquad s(N) \overset{\text{déf}}{=} \sum_{n=0}^{N} \frac{1}{16^n}\left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6}\right).$$

- Écrire une fonction qui, pour N donné, renvoie $s(N)$.
- Écrire un script qui calcule pour quelles valeurs de N on obtient une approximation de π aussi précise que celle fournie par la variable π du module `math`.

Méthode 9 : Approche Monte-Carlo

Il est possible d'approcher les premières décimales de π en utilisant une approche aléatoire.

Considérons un carré de côté 1 défini par les points $(0,0),(0,1),(1,0),(1,1)$. Traçons un cercle de rayon 1 centré à l'origine et prenons le quart de ce cercle situé dans le premier quadrant. L'aire du carré est égale à 1, tandis que l'aire du quart de cercle est $\frac{\pi}{4}$. Si l'on divise l'aire du quart de cercle par celle du carré, on obtient donc $\frac{\pi}{4}$. Si l'on choisit aléatoirement un point dans le carré, il y a une probabilité de $\frac{\pi}{4}$ que le point se trouve également dans le quart de cercle.

Considérons l'algorithme suivant pour estimer π : générons N couples $(x_k, y_k)_{k=0}^{N-1}$ de nombres aléatoires (avec une distribution uniforme) dans l'intervalle $[0,1]$, puis calculons le nombre $m \le N$ de ceux qui se trouvent dans le quart de cercle.

Notons $\tilde{\pi}(N) \overset{\text{déf}}{=} 4m/N$.

Écrire un programme pour calculer $\tilde{\pi}(N)$ et obserer comment l'erreur évolue avec l'augmentation de N, sachant que $\tilde{\pi}(N) \to \pi$ lorsque $N \to +\infty$.

Méthode 10: Intégrale double

On peut montrer que

$$\pi = \int_0^1 \left(\int_0^{\sqrt{1-x^2}} 4 \right) \mathrm{d}y\,\mathrm{d}x$$

Utiliser `scipy.integrate` pour approcher cette intégrale.

Méthode 11: Suite de Borwein

Considérons les suites $\{a_n\}_{n\in\mathbb{N}}$, $\{s_n\}_{n\in\mathbb{N}}$ et $\{r_n\}_{n\in\mathbb{N}}$ ainsi construites:

$$\begin{cases} s_0 = \frac{\sqrt{3}-1}{2}, \\ r_0 = \frac{3}{1+2\sqrt[3]{1-s_n^3}}, \\ a_0 = \frac{1}{3}; \end{cases} \qquad \begin{cases} r_{n+1} = \frac{3}{1+2\sqrt[3]{1-s_n^3}}, \\ s_{n+1} = \frac{r_{n+1}-1}{2}, \\ a_{n+1} = r_{n+1}^2 a_n - 3^n (r_{n+1}^2 - 1). \end{cases}$$

On peut montrer que

$$\pi = \lim_{n\to\infty} 1/a_n$$

Écrire un script qui affiche a_n pour $n = 0,\dots,3$ et calcule l'erreur.

Méthode 12: Suite de Jonathan-Borwein

Soit $f(x) = \sqrt[4]{1-x^4}$ et considérons les deux suites $\{y_n\}_{n\in\mathbb{N}}$, et $\{a_n\}_{n\in\mathbb{N}}$ ainsi construites:

$$\begin{cases} y_0 = \sqrt{2}-1, \\ a_0 = 6-4\sqrt{2}; \end{cases} \qquad \begin{cases} y_{n+1} = \frac{1-f(y_n)}{1+f(y_n)}, \\ a_{n+1} = a_n(1+y_{n+1})^4 - 2^{2n+3} y_{n+1}(1+y_{n+1}+y_{n+1}^2). \end{cases}$$

On peut montrer que

$$\pi = \lim_{n\to\infty} 1/a_n$$

Écrire un script qui affiche a_n pour $n = 0,\dots,3$ et calcule l'erreur.

Méthode 13: Suite de Gauss-Legendre

Considérons les quatre suites $\{a_n\}_{n\in\mathbb{N}}$, $\{b_n\}_{n\in\mathbb{N}}$,$\{p_n\}_{n\in\mathbb{N}}$, $\{t_n\}_{n\in\mathbb{N}}$ ainsi construites:

$$\begin{cases} a_0 = 1, \\ b_0 = 1/\sqrt{2}, \\ p_0 = 1, \\ t_0 = 1/4; \end{cases} \qquad \begin{cases} a_{n+1} = \frac{a_n+b_n}{2}, \\ b_{n+1} = \sqrt{a_n b_n}, \\ p_{n+1} = 2p_n, \\ t_{n+1} = t_n - p_n(a_n - a_{n+1})^2. \end{cases}$$

On peut montrer que

$$\pi = \lim_{n\to\infty} f(n) \qquad \text{avec} \qquad f(n) = \frac{(a_{n+1}+b_{n+1})^2}{4t_{n+1}}.$$

Écrire un script qui affiche $f(n)$ pour $n = 0,\dots,3$ et calcule l'erreur.

Méthode 14: Le module `decimal`

Les méthodes précédentes utilisent des nombres flottants pour toutes les variables et la somme partielle, ainsi la précision finale de l'approximation de π sera extrêmement limitée (et ne sera pas meilleure que d'importer `math.pi`). On peut essayer d'utiliser des nombres décimaux à la place, en important le module `decimal`. Comprendre le code suivant:

```
from decimal import *
mypi = Decimal (22) / Decimal (7)
print(mypi)
```

Lire la documentation du module pour améliorer la précision.

Source: https://en.wikipedia.org/wiki/Approximations_of_%CF%80

Correction

Pour une meilleur affichage, on pourra utiliser la fonction `tabulate` du module `tabulate`.

Méthode 1: Fractions

```
from tabulate import tabulate
T = []
T.append( ["Formule" , "Approximation", "Erreur"] )
```

```
from math import pi
for formule in [ "22/7" , "355/113", "3 + 8/60 + 29/60**2 + 44/60**3" ]:
    approx = eval(formule)
    T.append( [formule, approx, approx-pi] )

print(tabulate(T,headers="firstrow",floatfmt="0.9f"))
```

```
Formule                          Approximation        Erreur
------------------------------  ---------------  ------------
22/7                                3.142857143   0.001264489
355/113                             3.141592920   0.000000267
3 + 8/60 + 29/60**2 + 44/60**3      3.141592593  -0.000000061
```

Bonus : on peut calculer une primitive (et ensuite l'intégrale) analytiquement (à la main ou avec le module sympy) :

```
func = lambda x: x**4*(1-x)**4/(1+x**2)

# VALEURE EXACTE  DE L'INTEGRALE
import sympy as sp
x = sp.Symbol('x')
primitive = sp.integrate(func(x), x)
print(f"Une primitive est {primitive}")
integrale_exacte = primitive.subs(x, 1) - primitive.subs(x, 0)
print(f"donc l'integrale vaut {integrale_exacte}")


# BORNES DE L'ERREUR
funcLEFT  = lambda x: x**4*(1-x)**4/2
funcRIGHT = lambda x: x**4*(1-x)**4
primitiveLEFT = sp.integrate(funcLEFT(x), x)
integrale_exacteLEFT = primitiveLEFT.subs(x, 1) - primitiveLEFT.subs(x, 0)
primitiveRIGHT = sp.integrate(funcRIGHT(x), x)
integrale_exacteRIGHT = primitiveRIGHT.subs(x, 1) - primitiveRIGHT.subs(x, 0)
print(f"On a {integrale_exacteLEFT} < {integrale_exacte}  < {integrale_exacteRIGHT}")
print(f"En effet {integrale_exacteLEFT} =  {integrale_exacteLEFT.evalf()}")
print(f"         {integrale_exacte} ≈  {integrale_exacte.evalf()}")
print(f"         {integrale_exacteRIGHT} =  {integrale_exacteRIGHT.evalf()}")
```

```
Une primitive est x**7/7 - 2*x**6/3 + x**5 - 4*x**3/3 + 4*x - 4*atan(x)
donc l'integrale vaut 22/7 - pi
On a 1/1260 < 22/7 - pi  < 1/630
En effet 1/1260 =  0.000793650793650794
         22/7 - pi ≈  0.00126448926734962
         1/630 =  0.00158730158730159
```

Méthode 2 : Racines

```
from tabulate import tabulate
T = []
T.append( ["Formule" , "Approximation", "Erreur"] )

from math import pi, sqrt
for formule in [ "sqrt(2)+sqrt(3)" , "31**(1/3)", "sqrt(63)-sqrt(23)", "sqrt(51)-sqrt(16)",
 ↪  "28658146**(1/15)", "(2143/22)**(1/4)" ]:
    approx = eval(formule)
    T.append( [formule, approx, approx-pi] )

print(tabulate(T,headers="firstrow",floatfmt="0.12f"))
```

```
Formule              Approximation           Erreur
-----------------  ---------------  ---------------
sqrt(2)+sqrt(3)     3.146264369942   0.004671716352
31**(1/3)           3.141380652391  -0.000212001198
sqrt(63)-sqrt(23)   3.141422409881  -0.000170243709
sqrt(51)-sqrt(16)   3.141428428543  -0.000164225047
28658146**(1/15)    3.141592653814   0.000000000224
(2143/22)**(1/4)    3.141592652583  -0.000000001007
```

Méthode 3: Formule de Liu Hui

```
from math import sqrt
pi_approx = 768 * sqrt(2 - sqrt(2 + sqrt(2 + sqrt(2 + sqrt(2 + sqrt(2 + sqrt(2 + sqrt(2 +
↪  sqrt(2+1))))))))
print(f"pi_approx={pi_approx:0.6f}, erreur={pi_approx-pi:0.6f}")
```

```
pi_approx=3.141590, erreur=-0.000002
```

Méthode 4: Logarithmes

```
from math import pi, sqrt, log
formule = "log(640320**3+744)/sqrt(163)"
approx = eval(formule)
print(f"{approx=}\n{    pi=}")
formule = "12*log( (2*sqrt(2)+sqrt(10))*(3+sqrt(10)) )/sqrt(190)"
approx = eval(formule)
print(f"{approx=}\n{    pi=}")
```

```
approx=3.141592653589793
    pi=3.141592653589793
approx=3.1415926535897936
    pi=3.141592653589793
```

Méthode 5: Formule de Basel

```
u = lambda n : 1/n**2

from math import pi
n, somme = 0, 0
while abs(somme-pi**2/6)>1e-5:
    n += 1
    somme += u(n)
print(f"N={n}, somme={somme:0.6f}, pi^2/6={pi**2/6:0.6f}")
```

```
N=100000, somme=1.644924, pi^2/6=1.644934
```

Méthode 6: Formule de Madhava–Leibniz

```
u = lambda n : (-1)**n/(2*n+1)

from math import pi
n, somme = 0, 4
while abs(somme-pi)>1e-5:
    n += 1
    somme += 4*u(n)
print(f"N={n}, somme={somme:0.6f}, pi/4={pi:0.6f}")
```

```
N=100000, somme=3.141603, pi/4=3.141593
```

Méthode 7: Formule de Madhava

```
u = lambda n : (-3)**(-n)/(2*n+1)

from math import pi
n, somme = 0, (12)**(0.5)
while abs(somme-pi)>1e-5:
    n += 1
    somme += (12)**(0.5)*u(n)
print(f"N={n}, somme={somme:0.6f}, pi/4={pi:0.6f}")
```

```
N=8, somme=3.141600, pi/4=3.141593
```

Méthode 8: Formule de Bailey-Borwein-Plouffe (BBP)
On note

$$u(n) = 16^{-n}\left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6}\right) \qquad \pi_{\text{approché}}(\mathrm{N}) = \sum_{n=0}^{\mathrm{N}} u(n)$$

```
from math import pi

u = lambda n :  ( 4/(8*n+1) - 2/(8*n+4) - 1/(8*n+5) - 1/(8*n+6) )*(1/16)**n
pi_approche = lambda N :  sum( [ u(n) for n in range(N+1)] )

N = 0
```

```
while abs(pi-pi_approche(N))>0:
    N += 1

print(f"N = {N}")
print("pi approx =",pi_approche(N))
print(" math.pi  =",pi)
```

```
N = 10
pi approx = 3.141592653589793
 math.pi  = 3.141592653589793
```

Pour N = 10 on obtient une approximation de π qui coïncide (à la précision Python) avec la variable interne `math.pi`. Cet algorithme est en effet extrêmement efficace et permet le calcul rapide de centaines de chiffres significatifs de π.

Remarque : on peut améliorer cette implémentation en calculant, à chaque itération, juste $u(N+1)$ et en posant $\pi_{\text{approché}}(N+1) = \pi_{\text{approché}}(N) + u(N+1)$ (dans le script précédent on recalcule tous les termes de la suite $\{u(n)\}_n$ à chaque appelle à la fonction `pi_approx`).

Pour bien visualiser la convergence on affichera, pour chaque N, la valeur approchée et l'erreur. La mise en forme sera effectuée grâce à la fonction `tabulate` du module `tabulate` :

```
from tabulate import tabulate
table = []
table.append(["N", "Approximation", "Erreur"])

u = lambda n :  ( 4/(8*n+1) - 2/(8*n+4) - 1/(8*n+5) - 1/(8*n+6) )*(1/16)**n
N = 0
pi_approx = u(N)
from math import pi
table.append( [N, pi_approx, pi-pi_approx] )
while abs(pi-pi_approx)>0:
    N += 1
    pi_approx += u(N)
    table.append( [N, pi_approx, pi-pi_approx] )

print(tabulate(table,headers="firstrow",floatfmt=".15f"))
```

```
  N      Approximation             Erreur
---  -----------------  -----------------
  0  3.133333333333333  0.008259320256460
  1  3.141422466422466  0.000170187167327
  2  3.141587390346582  0.000005263243211
  3  3.141592457567436  0.000000196022357
  4  3.141592645460336  0.000000008129457
  5  3.141592653228088  0.000000000361705
  6  3.141592653572881  0.000000000016912
  7  3.141592653588973  0.000000000000820
  8  3.141592653589752  0.000000000000041
  9  3.141592653589791  0.000000000000002
 10  3.141592653589793  0.000000000000000
```

Méthode 9 : Approche Monte-Carlo

```
import random

def tilde_pi(N): # On tirera N points
    m = 0     # Compteur pour les points qui tombent dans le disque
    for k in range(N):
        # On tire au hasard un point [x,y] dans [0,1[ x [0,1[
        P = [ random.uniform(0,1), random.uniform(0,1) ]
        m += (P[0]**2+P[1]**2<=1) # Si ==True alors le point est dans le disque
    return 4*m/N

# TEST
from math import pi
N = 1000
myPi = tilde_pi(N) # notre approximation de pi
err = abs(pi-myPi) # on compare à la valeur de pi donnée par le module math
print(f"{pi = :g}, {myPi = :g}, erreur = {err:g}")
```

Pour N = 1000 on obtient

```
pi = 3.14159, myPi = 3.184, erreur = 0.0424073
```

Naturellement, comme les nombres sont générés aléatoirement, les résultats obtenus pour une même valeur de N peuvent changer à chaque exécution.

Plus N est grand, meilleure est l'approximation de π. Cependant, cette méthode n'est pas très efficace, il faut beaucoup de tirs pour obtenir le deux premières décimales de π (pour obtenir chaque nouvelle décimale, nous devons augmenter le nombre de points aléatoires d'un facteur 10).

Méthode 10: Intégrale double

```
# VALEURE EXACTE  DE L'INTEGRALE
import sympy as sp
x = sp.Symbol('x')
y = sp.Symbol('y')
exacte = sp.integrate(  sp.integrate(4, (y,0,sp.sqrt(1-x**2))) , (x,0,1) )
print(f"L'intégrale vaut exactement {exacte }")

# VALEURE APPROCHÉE   DE L'INTEGRALE
from scipy.integrate import quad
from numpy import sqrt
approx = quad(lambda x: quad(lambda y: 4, 0, sqrt(1-x**2))[0], 0, 1)[0]
print(f"L'integrale vaut approximativement {approx = }")

from numpy import sqrt
from scipy.integrate import dblquad
# NB l'ordre des variables dans la fonction lambda : d'abord l'intégrale interne
# ∫_a^b (∫_c^{b(x)} f(x,y) dy) dx dblquad(lambda y, x: f(x,y), a, b, lambda x: c, lambda x: b(x))
approx = dblquad(lambda y, x: 4, 0, 1, lambda x: 0, lambda x: sqrt(1-x**2))[0]
print(f"L'integrale vaut approximativement {approx = }")
```

```
L'intégrale vaut exactement pi
L'integrale vaut approximativement approx = 3.1415926535897922
L'integrale vaut approximativement approx = 3.1415926535897922
```

Méthode 11: Suite de Borwein

```
from math import pi,sqrt
s,a = (sqrt(3)-1)/2 , 1/3
for n in range(4):
    r = 3/(1+2*(1-s**3)**(1/3))
    s = (r-1)/2
    a = r**2*a-3**n*(r**2-1)
    pi_approx = 1/a
    print(f"{n=}, {pi_approx=:0.15f}, erreur={pi_approx-pi:0.15f}")
```

```
n=0, pi_approx=3.141590585205897, erreur=-0.000002068383896
n=1, pi_approx=3.141592653589804, erreur=0.000000000000011
n=2, pi_approx=3.141592653589804, erreur=0.000000000000011
n=3, pi_approx=3.141592653589804, erreur=0.000000000000011
```

Méthode 12: Suite de Jonathan-Borwein

```
from math import pi,sqrt
f = lambda x : (1-x**4)**(1/4)
y,a = sqrt(2)-1, 6-4*sqrt(2)
for n in range(4):
    y = (1-f(y))/(1+f(y))
    a = a*(1+y)**4-2**(2*n+3)*y*(1+y+y**2)
    pi_approx = 1/a
    print(f"{n=}, {pi_approx=:0.15f}, erreur={pi_approx-pi:0.15f}")
```

```
n=0, pi_approx=3.141592646213547, erreur=-0.000000007376246
n=1, pi_approx=3.141592653589793, erreur=-0.000000000000000
n=2, pi_approx=3.141592653589793, erreur=-0.000000000000000
n=3, pi_approx=3.141592653589793, erreur=-0.000000000000000
```

Méthode 13: Suite de Gauss-Legendre

```
from math import pi, sqrt
a,b,t,p = 1, 1/sqrt(2), 1/4, 1
```

```
for n in range(4):
    a_new = (a+b)/2
    a,b,t,p = a_new, sqrt(a*b), t-p*(a-a_new)**2, 2*p
    pi_approx = (a+b)**2/(4*t)
    print(f"{n=}, {pi_approx=:0.15f}, erreur={pi_approx-pi:0.15f}")
```

```
n=0, pi_approx=3.140579250522169, erreur=-0.001013403067625
n=1, pi_approx=3.141592646213543, erreur=-0.000000007376250
n=2, pi_approx=3.141592653589794, erreur=0.000000000000001
n=3, pi_approx=3.141592653589794, erreur=0.000000000000001
```

Méthode 14: Le module `decimal`

```
from decimal import *
mypi = Decimal (22) / Decimal (7)
print(mypi)
```

```
3.142857142857142857142857143
```

https://docs.python.org/fr/3/library/decimal.html

```
from decimal import *
getcontext().prec = 50
mypi = Decimal (22) / Decimal (7)
print(mypi)
```

```
3.14285714285714285714285714285714285714285714285 71
```

Exercice 7.46 (Module `scipy` - Calcul approché d'une intégrale (méthode Monte-Carlo))

La méthode de Monte-Carlo (du nom des casinos, pas d'une personne) est une approche probabiliste permettant d'**approcher la valeur de l'intégrale**

$$\mathrm{J} = \int_a^b f(x)\,\mathrm{d}x.$$

L'idée de base est que l'intégrale J peut être vue comme l'espérance d'une variable aléatoire uniforme X sur l'intervalle $[a, b]$. Par la loi des grands nombres cette espérance peut être approchée par la moyenne empirique

$$\mathrm{J} \approx \mathrm{J}(\mathrm{N}) = \frac{b-a}{\mathrm{N}} \sum_{i=0}^{\mathrm{N}-1} f(x_i),$$

où les x_i sont tirés aléatoirement dans l'intervalle $[a, b]$ avec une loi de probabilité uniforme.

- Écrire une fonction `montecarlo(f,a,b,N)` qui calcule J_N.
- Valider la fonction (*i.e.* on considère des cas dont on connaît la solution exacte et on écrit un test unitaire). Quelle valeur obtient-on avec le module `scipy.integrate`?

Correction

```
import random
montecarlo = lambda f,a,b,N : (b-a)*sum([f(random.uniform(a,b)) for i in range(N)])/N

# TESTS
from scipy import integrate

g = lambda x: 1
print("Calcul approché par Montecarlo =",montecarlo(g,0,1,100))
print("Calcul approché par ScyPy =", integrate.quad(g,0,1)[0])

g = lambda x: x**3
print("Calcul approché par Montecarlo =",montecarlo(g,0,1,100))
print("Calcul approché par ScyPy =", integrate.quad(g,0,1)[0])
```

```
Calcul approché par Montecarlo = 1.0
Calcul approché par ScyPy = 1.0
Calcul approché par Montecarlo = 0.2341070188028165
Calcul approché par ScyPy = 0.25
```

Exercice 7.47 (Module `numpy` - Statistique)

Soit L une liste de n valeurs. On rappelle les définitions suivantes :

Moyenne arithmétique $\overline{m} = \dfrac{\sum L_i}{n}$

Variance $V = \dfrac{\sum (L_i - \overline{m})^2}{n-1}$

Écart-type $\sigma = \sqrt{V}$

Médiane C'est la valeur qui divise l'échantillon en deux parties d'effectifs égaux. Soit C la liste qui contient les composantes de L ordonnées, alors

$$\text{médiane} = \begin{cases} \dfrac{C_{\frac{n}{2}} + C_{\frac{n}{2}+1}}{2} & \text{si } n \text{ est pair,} \\ C_{\frac{n}{2}+1} & \text{si } n \text{ est impair.} \end{cases}$$

Attention, dans cette définition les indices commencent à 1.

Écrire trois fonctions qui renvoient respectivement la moyenne, l'écart type et la médiane d'une liste donnée en entré. Vérifier l'implémentation sur deux listes bien choisies et comparer avec les calculs effectués par les fonctions `mean`, `std` et `median` prédéfinies dans le module `numpy` (à importer).

Correction

Pour vérifier l'implémentation, il faut pouvoir tester les deux cas possibles pour le calcul de la médiane, à savoir une liste qui a un nombre pair d'éléments puis une liste qui en a un nombre impair.

```
from numpy import *

moyenne = lambda L : sum(L)/len(L)

def variance(L):
    m = moyenne(L)
    return sum((ell-m)**2 for ell in L)/(len(L)-1) # cas d'un échantillon

ecart_type = lambda L : (variance(L))**0.5

def mediane(L):
    C = sorted(L)
    n = len(L)
    if n%2==0 :
        return (C[n//2-1]+C[n//2])/2
    else:
        return C[n//2]

# TESTS
for L in ( [0, 0, 8, 1, 1, 2, 2] , [2, 3, 3, 2] ):
    print(f"{L = }")
    print(f"{moyenne(L) = }, avec numpy on trouve {mean(L) = }")
    print(f"{ecart_type(L) = }, avec numpy on trouve {std(L,ddof=1) = }")
    print(f"{mediane(L) = }, avec numpy on trouve {median(L) = }\n")
```

```
L = [0, 0, 8, 1, 1, 2, 2]
moyenne(L) = 2.0, avec numpy on trouve mean(L) = 2.0
ecart_type(L) = 2.7688746209726918, avec numpy on trouve std(L,ddof=1) = 2.7688746209726918
mediane(L) = 1, avec numpy on trouve median(L) = 1.0

L = [2, 3, 3, 2]
moyenne(L) = 2.5, avec numpy on trouve mean(L) = 2.5
ecart_type(L) = 0.5773502691896257, avec numpy on trouve std(L,ddof=1) = 0.5773502691896257
mediane(L) = 2.5, avec numpy on trouve median(L) = 2.5
```

Exercice 7.48 (Module `numpy` - Statistique et suites)

Soit $\mathbf{x}$ une liste de $n \in \mathbb{N}^*$ valeurs et définissons la fonction $\mathrm{F}(\mathbf{x}) = (\mathrm{mean}(\mathbf{x}), \mathrm{geometric\_mean}(\mathbf{x}), \mathrm{median}(\mathbf{x}))$.

On veut vérifier que, quel que soit la liste $\mathbf{x}$, il existe une valeur ℓ telle que $\mathrm{F}(\mathrm{F}(\mathrm{F}(\ldots(\mathrm{F}(\mathbf{x}))))) = (\ell, \ell, \ell)$, c'est-à-dire que la suite $\{\mathbf{y}_n\}_n$ définie par récurrence

$$\begin{cases} \mathbf{y}_0 = \mathrm{F}(\mathbf{x}) \\ \mathbf{y}_{n+1} = \mathrm{F}(\mathbf{y}_n) \end{cases}$$

converge vers un triplet (ℓ, ℓ, ℓ).

On pourra utiliser les fonctions `mean`, `geometric_mean`, `median` du module `statistics`.

Correction

Pour une meilleur affichage, on utilise le module `tabulate`.

```
from tabulate import tabulate
T = []
T.append(["i","yy[0]","yy[1]","yy[2]"])

from statistics import mean, geometric_mean, median
f = lambda xx : (mean(xx),geometric_mean(xx),median(xx))

from random import *
n = randint(20,50)
xx = [randint(1,100) for _ in range(n)]

i = 0
yy = f(xx)
T.append([f"{i}",f"{yy[0]}",f"{yy[1]}",f"{yy[2]}"])

while abs(yy[1]-yy[0])>1.e-12 or abs(yy[1]-yy[2])>1.e-12:
    yy = f(yy)
    i += 1
    T.append([f"{i}",f"{yy[0]}",f"{yy[1]}",f"{yy[2]}"])
print(tabulate(T,headers="firstrow",floatfmt=".13f"))
```

```
  i             yy[0]             yy[1]             yy[2]
---  ----------------  ----------------  ----------------
  0  48.9705882352941  32.4621606655858  49.0000000000000
  1  43.4775829669600  42.7073811765945  48.9705882352941
  2  45.0518507929495  44.9677562507911  43.4775829669600
  3  44.4990633369002  44.4931427095082  44.9677562507911
  4  44.6533207657332  44.6527684518777  44.4990633369002
  5  44.6017175181704  44.6016584056433  44.6527684518777
  6  44.6187147918971  44.6187082959790  44.6017175181704
  7  44.6130468686822  44.6130461493575  44.6187082959790
  8  44.6149337713395  44.6149336915084  44.6130468686822
  9  44.6143047771767  44.6143047683099  44.6149336915084
 10  44.6145144123317  44.6145144113466  44.6143047771767
 11  44.6144445336183  44.6144445335089  44.6145144113466
 12  44.6144678261579  44.6144678261458  44.6144445336183
 13  44.6144600619740  44.6144600619727  44.6144678261458
 14  44.6144626500308  44.6144626500307  44.6144600619740
 15  44.6144617873452  44.6144617873451  44.6144626500307
 16  44.6144620749070  44.6144620749070  44.6144617873452
 17  44.6144619790530  44.6144619790531  44.6144620749070
 18  44.6144620110044  44.6144620110044  44.6144619790531
 19  44.6144620003539  44.6144620003539  44.6144620110044
 20  44.6144620039041  44.6144620039041  44.6144620003539
 21  44.6144620027207  44.6144620027207  44.6144620039041
 22  44.6144620031152  44.6144620031152  44.6144620027207
 23  44.6144620029837  44.6144620029837  44.6144620031152
 24  44.6144620030275  44.6144620030275  44.6144620029837
 25  44.6144620030129  44.6144620030129  44.6144620030275
 26  44.6144620030178  44.6144620030178  44.6144620030129
```

```
27  44.6144620030162  44.6144620030162  44.6144620030178
28  44.6144620030167  44.6144620030167  44.6144620030162
```

Exercice Bonus 7.49 (Module `numpy` - Représentation et manipulation de polynômes)
Résoudre l'exercice 6.78 en utilisant le sous-module `polynomial` du module `numpy`.

Correction
Première méthode :

```
import numpy as np
from numpy.polynomial import polynomial

p = np.array([1, 2, 3])
x = np.array([-1,0,1,2])
y = polynomial.polyval(x,p)
print( f"p(x)={p} => p({x})={y}" )

p = np.array([1, 2, 3])
q = np.array([1, -2])
somme = polynomial.polyadd(p,q)
print(f"p(x)={p}, q(x)={q} => (p+q)(x)={somme}")

p = np.array([1, 0, 3])
q = np.array([1, -2])
mul = polynomial.polymul(p,q)
print(f"p(x)={p}, q(x)={q} => (pq)(x)={mul}")

p = np.array([1, 2, 6])
d = polynomial.polyder(p)
print(f"p(x)={p} => p'(x)={d}")

p = np.array([1, 2, 6])
q = polynomial.polyint(p)
print(f"q'(x)={p} => q(x)={q}")
```

```
p(x)=[1 2 3] => p([-1  0  1  2])=[ 2.  1.  6. 17.]
p(x)=[1 2 3], q(x)=[ 1 -2] => (p+q)(x)=[2. 0. 3.]
p(x)=[1 0 3], q(x)=[ 1 -2] => (pq)(x)=[ 1. -2.  3. -6.]
p(x)=[1 2 6] => p'(x)=[ 2. 12.]
q'(x)=[1 2 6] => q(x)=[0. 1. 1. 2.]
```

Deuxième méthode

```
from numpy.polynomial import Polynomial

p = Polynomial([1, 2, 3])
x = np.array([-1,0,1,2])
y = p(x)
print( f"p(x)={p} => p({x})={y}" )

p = Polynomial([1, 0, 3])
q = Polynomial([1, -2])
s = p+q
print(f"p(x)={p}, q(x)={q} => (p+q)(x)={s}")

p = Polynomial([1, 0, 3])
q = Polynomial([1, -2])
m = p*q
print(f"p(x)={p}, q(x)={q} => (pq)(x)={m}")

p = Polynomial([1, 2, 6])
d = p.deriv()
print(f"p(x)={p} => p'(x)={d}")
```

```
p = Polynomial([1, 2, 6])
q = p.integ()
print(f"q'(x)={p} => q(x)={q}")
```

```
p(x)=1.0 + 2.0·x + 3.0·x² => p([-1  0  1  2])=[ 2.  1.  6. 17.]
p(x)=1.0 + 0.0·x + 3.0·x², q(x)=1.0 - 2.0·x => (p+q)(x)=2.0 - 2.0·x + 3.0·x²
p(x)=1.0 + 0.0·x + 3.0·x², q(x)=1.0 - 2.0·x => (pq)(x)=1.0 - 2.0·x + 3.0·x² - 6.0·x³
p(x)=1.0 + 2.0·x + 6.0·x² => p'(x)=2.0 + 12.0·x
q'(x)=1.0 + 2.0·x + 6.0·x² => q(x)=0.0 + 1.0·x + 1.0·x² + 2.0·x³
```

Exercice Bonus 7.50 (PyDéfis – Les victimes de Tooms 1/2)

Contexte: Eugene W. Tooms, un criminel notoire, a probablement fait plus de victimes que ce que l'on croit. En fouillant dans d'anciens dossiers, l'agent Mulder a retrouvé des relevés de morsures et a soumis Tooms à un test pour relever ses empreintes dentaires. Il souhaite maintenant identifier quelles victimes parmi les meurtres non élucidés correspondent à Tooms.

Problème: chaque relevé de morsure est composé de 10 nombres, représentant un profil dentaire. Le relevé de Tooms est 10, 12, 6, 9, 18.5, 22, 7, 4, 9, 10. Lorsqu'il produit une morsure, si elle est peu profonde, chaque valeur est diminuée d'une même quantité, si elle est profonde, chaque valeur est augmentée d'une même quantité.

Objectif: le fichier d'entrée contient un relevé par ligne. Chaque ligne commence par le numéro du relevé, suivi des 10 valeurs numériques du relevé. Recherchez tous les numéros de relevés qui correspondent à Tooms. Pour valider le défi, additionnez ces numéros de relevés.

EXEMPLE DE RELEVÉS :

- 8, 10, 4, 7, 16.5, 20, 5, 2, 7, 8 (diminution de 2 : morsure peu profonde)
- 11.5, 13.5, 7.5, 10.5, 20, 23.5, 8.5, 5.5, 10.5, 11.5 (augmentation de 1.5 : morsure profonde)
- 11.5, 10, 7.5, 10.5, 20, 20, 8.5, 5.5, 7, 11.5 (relevé ne correspondant pas à Tooms)

Si le fichier d'entrée contient

```
001 - 15.25, 16.25, 11.25, 14.25, 22.75, 26.25, 11.25, 8.25, 14.25, 15.25
002 - 10.75, 12.75, 6.75, 6.0, 15.5, 22.75, 4.0, 4.75, 6.0, 7.0
003 - 11.5, 13.5, 7.5, 10.5, 20.0, 23.5, 8.5, 5.5, 10.5, 11.5
004 - 14.25, 16.25, 10.25, 5.0, 14.5, 26.25, 3.0, 8.25, 5.0, 6.0
005 - 12.25, 14.25, 8.25, 11.25, 20.75, 24.25, 9.25, 6.25, 11.25, 12.25
006 - 12.75, 11.75, 5.75, 8.75, 21.25, 24.75, 9.75, 3.75, 11.75, 9.75
007 - 11.0, 13.0, 7.0, 10.0, 19.5, 23.0, 8.0, 5.0, 10.0, 11.0
```

Les relevés correspondant à Tooms seront 003, 005, 007. La somme des numéros de relevés est $3+5+7=15$.

Source : https://pydefis.callicode.fr/defis/C24_DentToomsEasy/txt

Exercice Bonus 7.51 (Module `sympy` - Représentation et manipulation de polynômes)

Résoudre l'exercice 6.78 en utilisant le module `sympy`.

Correction

```
import sympy
x = sympy.symbols('x')

p = 1+2*x+6*x**2
q = 1-2*x

xx = [-1,0,1,2]
yy = [p.subs(x,xi) for xi in xx]
print(f"p(x)={p} => p({xx})={yy}")

somme = p+q
mul = p*q
print(f"p(x)={p}, q(x)={q} => (p+q)(x)={somme}")
print(f"p(x)={p}, q(x)={q} => (pq)(x)={mul}")

d = p.diff()
```

```
i = p.integrate()
print(f"p(x)={p} => d(x)=p'(x)={d}")
print(f"i'(x)=p(x)={p} => i(x)={i}")
```

```
p(x)=6*x**2 + 2*x + 1 => p([-1, 0, 1, 2])=[5, 1, 9, 29]
p(x)=6*x**2 + 2*x + 1, q(x)=1 - 2*x => (p+q)(x)=6*x**2 + 2
p(x)=6*x**2 + 2*x + 1, q(x)=1 - 2*x => (pq)(x)=(1 - 2*x)*(6*x**2 + 2*x + 1)
p(x)=6*x**2 + 2*x + 1 => d(x)=p'(x)=12*x + 2
i'(x)=p(x)=6*x**2 + 2*x + 1 => i(x)=2*x**3 + x**2 + x
```

Exercice Bonus 7.52 (Permutations avec itertools)
Soit un nombre à 10 chiffres tous différents. On pourra écrire ce nombre sous la forme $\underline{abcdefghij}$ ayant noté avec une barre la suite des chiffres qui le composent dans l'écriture décimale. Trouver le seul nombre tel que $\underline{a}$ est divisible par 1, $\underline{ab}$ est divisible par 2, $\underline{abc}$ est divisible par 3, etc. $\underline{abcdefghij}$ est divisible par 10.

Correction
On considère les permutations de la liste de chaînes de caractères `C = ["0","1","2","3","4","5","6","7","8","9"]`. La concatenation des éléments d'une permutation de `C` donne un nombre à 10 chiffres tous différents. Pour chaque liste `P` qui est une permutation de la liste `C`, on vérifie toutes les conditions. Dès qu'elles sont toutes satisfaites, on sort de la fonction.

```
import itertools
def f():
    C = [str(i) for i in range(10)]
    for P in list(itertools.permutations(C)):
        cond = [ int(''.join(P[:j]))%j==0   for j in range(1,11) ]
        if all(cond):
            return int(''.join(P))

print(f())
```

```
3816547290
```

Une autre stratégie consiste à tester tous les cas possibles en aidant la construction de ce nombre :
- $j = 0$ car $\underline{abcdefghij}$ est divisible par 10,
- $e = 5$ car $\underline{abcde}$ est divisible par 5 et 0 a été déjà utilisé,
- $b, d, f, h \in 2, 4, 6, 8$ car $\underline{ab}$, $\underline{abcd}$, $\underline{abcdef}$ et $\underline{abcdefgh}$ sont tous divisibles par 2,
- $a, c, g, i \in \{1, 3, 7, 9\}$ car les autres chiffres ont été utilisés,

Notons que le code devient presque illisible.

```
def f():
    j = 0
    e = 5
    for a in [1,3,7,9]:
        for b in range(2,9,2):
            for c in [1,3,7,9]:
                abc = int(str(a)+str(b)+str(c))
                if abc%3==0:
                    for d in range(2,9,2):
                        abcd = int(str(abc)+str(d))
                        if d!=b and abcd%4==0 :
                            for f in range(2,9,2):
                                abcdef = int(str(abcd)+str(5)+str(f))
                                if f!=b and f !=d and abcdef%6==0:
                                    for g in [1,3,7,9]:
                                        abcdefg = int(str(abcdef)+str(g))
                                        if g!=c and abcdefg%7==0:
                                            for h in range(2,9,2):
                                                abcdefgh = int(str(abcdefg)+str(h))
                                                if h!=b and h!=d and h!=f and abcdefgh%8==0:
                                                    for i in [1,3,7,9]:
                                                        abcdefghi = int(str(abcdefgh)+str(i))
```

```
    if i!=a and i!=c and i!=5 and i!=g and
    ↪ abcdefghi%9==0:
        return int(str(abcdefghi)+str(j))

print(f())
```

Exercice Bonus 7.53 (Permutations avec itertools)

Lors du premier tome de Harry Potter les trois héros doivent résoudre une énigme afin d'accéder à la salle où est cachée la pierre philosophale. Ce problème, dont l'auteur serait le professeur Rogue, consiste à trouver deux potions parmi les sept qui se trouvent devant eux: celles permettent d'avancer et de reculer. Ils sont aidés de quelques indices:

- Il y a trois fioles de poison, deux fioles de vin d'ortie, une fiole permettant d'avancer et une fiole permettant de reculer.
- Immédiatement à gauche de chacune des deux fioles de vin se trouve une fiole de poison.
- La première et la dernière fiole ont des contenus différents; ni l'une ni l'autre n'est la fiole qui permet d'avancer.
- Les contenus des fioles en position 1 et 5 sont identiques et ne contient pas du poison.

Trouver le contenu des 7 fioles. Pour générer toutes les permutations possibles, on pourra utiliser le module `itertools`. Voici un exemple d'utilisation

```
import itertools
for L in list(itertools.permutations(["A","B","C"])):
    print(L)
```

```
('A', 'B', 'C')
('A', 'C', 'B')
('B', 'A', 'C')
('B', 'C', 'A')
('C', 'A', 'B')
('C', 'B', 'A')
```

Source: http://www.xavierdupre.fr/app/ensae_teaching_cs/helpsphinx/specials/hermionne.html

Correction

On considère les permutations de la liste `F=["p","p","p","v","v","a","r"]` où "p" indique une fiole qui contient du poison, "v" du vin d'ortie, "a" si elle permet d'avancer et "r" de reculer. Pour chaque liste L qui est une permutation de la liste F, on vérifie toutes les conditions. Dès qu'elles sont toutes satisfaites, on sort de la fonction (sans continuer dans la boucle car on cherche un cas qui marche, pas tous les cas).

```
import itertools
def f():
    F = ["p","p","p","v","v","a","r"]
    for L in list(itertools.permutations(F)):
        if L[0]!=L[6]  and L[6]!="a" and L[0]!="a" and L[5]!="p" and L[2]!="p" and L[1]==L[5]:
            idx = [ i for i in range(1,7) if L[i]=="v"]
            if sum([L[i]=="v" and L[i-1]=="p" for i in idx])==2:
                return L

print(f())
```

```
('p', 'v', 'a', 'p', 'p', 'v', 'r')
```

Exercice Bonus 7.54 (Pydéfis – Analyse de séquences 1/2)

Il est courant pour Mulder et Scully (surtout pour Scully) de devoir classer d'anciens dossiers, et ces dossiers contiennent souvent des relevés de séquences d'acides nucléiques afin de détecter d'éventuelles mutations ou la présence de gènes extraterrestres.

Les relevés auxquels Scully s'intéresse proviennent d'ARN, décrit par une séquence des 4 bases nucléiques: A, C, G ou U (Adénine, Cytosine, Guanine, Uracile). Les séquences d'acides nucléiques sont parfois décrites à l'aide d'autres symboles, comme R (signifie A ou G), Y (signifie C ou U), B (signifie C, G ou U).

Voici la liste complète des symboles:

A signifie: A

C signifie: C

G signifie : G
R signifie : A ou G
K signifie : G ou U
S signifie : C ou G
B signifie : différent de A (c.-à-d. C, G ou U)
H signifie : différent de G (c.-à-d. A, C ou U)
N signifie : A, C, G ou U
U signifie : U
Y signifie : C ou U
M signifie : A ou C
W signifie : A ou U
D signifie : différent de C (c.-à-d. A, G ou U)
V signifie : différent de U (c.-à-d. A, C ou G)

La séquence ACDMR signifie donc :

- le premier acide nucléique est forcément A
- le deuxième acide nucléique est forcément C
- le troisième acide nucléique est A, G ou U
- le quatrième acide nucléique est A ou C
- le cinquième est A ou G

Il y a donc 12 séquences possibles qui correspondent à la description ACDMR : ACAAA, ACAAG, ACACA, ACACG, ACGAA, ACGAG, ACGCA, ACGCG, ACUAA, ACUAG, ACUCA, ACUCG.

Étant donné une séquence décrivant une séquence ARN avec certaines des 15 lettres du tableau précédent, Scully souhaite classer les dossiers en fonction du nombre de séquences ARN qui correspondent à cette description, ce qui lui permettra de répertorier d'une nouvelle manière les différents relevés trouvés dans les affaires non classées.

Cependant, la plupart des séquences sont assez longues, et le nombre correspondant est souvent très grand. Scully décide donc de ne conserver que les 5 derniers chiffres pour le classement.

La séquence BBBBDDDDDHHHH correspond à 531441 séquences différentes. Elle sera donc classée à 31441.

Aidez Scully en lui développant un système qui fait le travail demandé. Par exemple, pour la séquence BBBBDDDDHHHH, le système doit répondre : 31441.

Source : https://pydefis.callicode.fr/defis/C24_AcidesNucleiques_01/txt

Exercice Bonus 7.55 (Module `sympy` – devine le résultat)

Tester les codes suivants :

1. ```
   from sympy import *
   var('x')
   expr=(x**3+x**2-x-1)/(x**2+2*x+1)
   expr2=simplify(expr)
   print(expr,"=",expr2)
   ```
2. ```
   from sympy import *
   var('x')
   expr=x**4/2+5*x**3/12-x**2/3
   expr2=factor(expr)
   print(expr,"=",expr2)
   ```
3. ```
   from sympy import *
   var('x')
   expr=x**4-1
   sol=solve(expr)
   print(sol)
   ```
4. ```
   from sympy import *
   var('x,a,b,c')
   expr = a*x**2 + b*x + c
   sol=solve(expr, x)
   print(sol)
   ```
5. ```
   from sympy import *
   var('x,y')
   print(limit(cos(x),x,0))
   print(limit(x,x,oo))
   print(limit(1/x,x,oo))
   print(limit(x**x,x,0))
   print(diff(x*y**2,y))
   print(integrate(sin(x),x))
   ```

Correction

1. $\dfrac{x^3+x^2-x-1}{x^2+2x+1} = x-1$
2. $\dfrac{x^4}{2}+\dfrac{5x^3}{12}-\dfrac{x^2}{3} = \dfrac{x^2\cdot(2x-1)(3x+4)}{12}$
3. $[-1,\ 1,\ -i,\ i]$
4. $\left[\dfrac{-b-\sqrt{-4ac+b^2}}{2a},\ \dfrac{-b+\sqrt{-4ac+b^2}}{2a}\right]$
5. $1, \infty, 0, 1, 2xy, -\cos(x)$
   ```

Exercice Bonus 7.56 (Module `sympy` – calcul formel d'une dérivée)
Calculer $\partial_x f(x,y)$ pour
$$f(x,y) = \frac{ay-1}{(bx-1)^c}.$$

Correction

$$-\frac{bc\left(ay-1\right)(bx-1)^{-c}}{bx-1}$$

```
from sympy import *
var('x,y,a,b,c')
f=(a*y-1)/(b*x-1)**c
g=diff(f,x,1)
print(g)
```

Exercice Bonus 7.57 (Module `sympy` – composition de fonctions)
Calculer $f(m, f(n,m))$ si
$$f(a,b) = a^2 - b.$$

Correction

$$m^2 + m - n^2$$

```
import sympy as sym
sym.var('a,b,n,m')
f = lambda a,b :a**2-b
print( f(m,f(n,m)) )
```

Exercice Bonus 7.58 (Module `sympy` – calcul des paramètres)
La courbe d'équation $y = ax^2 + bx + c$ passe par le point $(1;\mathrm{P}_1)$ et la droite tangente à son graphe au point $(2;\mathrm{P}_2)$ a pente égale à 1. Calculer a, b, c.

Correction

```
import sympy as sym
sym.var('x,a,b,c,P_1,P_2')
f = a*x**2+b*x+c
eq1 = sym.Eq(f.subs(x,1),P_1)
eq2 = sym.Eq(f.subs(x,2),P_2)
fp = sym.diff(f,x)
eq3 = sym.Eq(fp.subs(x,2),1)
sol = sym.solve([eq1,eq2,eq3],[a,b,c])
print( sol )
```

$$\{a : \mathrm{P}_1 - \mathrm{P}_2 + 1,\ b : -4\mathrm{P}_1 + 4\mathrm{P}_2 - 3,\ c : 4\mathrm{P}_1 - 3\mathrm{P}_2 + 2\}$$

Exercice Bonus 7.59 (Module `sympy` – calcul des paramètres)
Soit $f : \mathbb{R} \to \mathbb{R}$ la fonction définie par $f(x) = \dfrac{ax+b}{x^2+1}$. Calculer a, b, c si f a un maximum local en $x_\mathrm{M} = 1$ et $f(x_\mathrm{M}) = c$.

Correction

$$\{a : 2c,\ b : 0\}$$

```
import sympy as sym
sym.var('x,a,b,c')
f=(a*x+b)/(x**2+1)
eq1=f.subs(x,1)-c
fp=sym.diff(f,x)
eq3=fp.subs(x,1)
sol=sym.solve([eq1,eq3],[a,b])
print( sol )
```

Exercice Bonus 7.60 (Module `sympy` – calcul de paramètres)
Soit $f : \mathbb{R} \to \mathbb{R}$ la fonction définie par $f(x) = x^4 - ax^3 + bx^2 + cx + d$. Si $b = -3a^2$, sur quel intervalle cette fonction est concave ?

Correction

$$\left[-\frac{a}{2}, a\right]$$

```
import sympy as sym
sym.var('x,a,b,c,d')
b=-3*a**2
f=x**4-a*x**3+b*x**2+c*x+d
fp=sym.diff(f,x)
fs=sym.diff(fp,x)
sol=sym.solve(fs,x)
print( sol )
```

CHAPITRE 8

Introduction à Matplotlib pour les tracés de base

La visualisation des données est un thème central dans *la science des données et le calcul scientifique*. `Matplotlib` est **l'un des nombreux packages de visualisation** disponibles en Python, mais il est considéré comme la référence. Son sous-package `pyplot`, généralement importé sous l'alias `plt`, est essentiel pour créer des visualisations.

EXEMPLE

Prenons un exemple concret : examinons l'évolution de la population mondiale. Nous disposons d'une liste d'années, `year`, et d'une liste de nombres d'habitants correspondantes, exprimées en milliards, `pop`. Par exemple, en 1970, environ 3.7 milliards de personnes vivaient sur Terre.

```
import matplotlib.pyplot as plt
year = [1950, 1970, 1990, 2010]
pop = [2.519, 3.692, 5.263, 6.972]
```

◇

Pour représenter ces données sous forme de points, nous utilisons la fonction `plt.scatter` en utilisant nos deux listes comme arguments :

```
plt.scatter(year, pop)
```

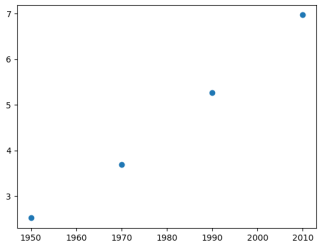
Python ne dessinera pas immédiatement le graphique. Il attendra l'appel de la fonction `plt.show` pour l'afficher :

```
# plt.show()
```

On peut bien sûr sauvegarder l'image :

```
plt.savefig('Images/populationScatter.png')
```

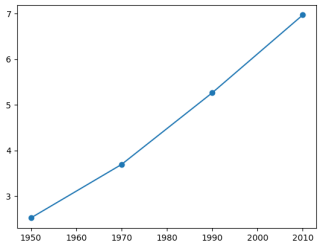
En examinant le graphique, nous constatons que les années sont représentées sur l'axe horizontal et la population correspondante sur l'axe vertical :



◇

Pour représenter ces données sous la forme d'une courbe (affine par morceaux), nous utilisons la fonction `plt.plot` en utilisant les mêmes arguments :

```
plt.plot(year, pop, '*-')
# plt.show()
plt.savefig('Images/population.png')
```



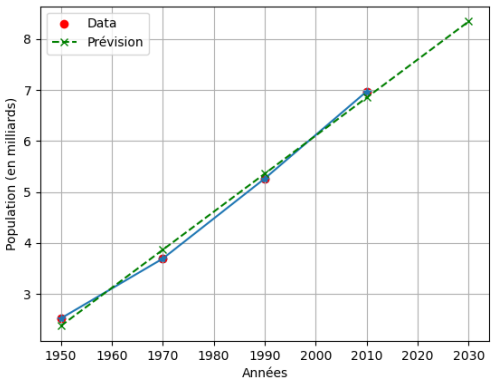
◇

En 1950, la population mondiale était d'environ 2.5 milliards. En 2010, elle était d'environ 7 milliards. Ainsi, la population mondiale a presque triplé en soixante ans. Si cette croissance se maintient, quel sera l'impact sur la surpopulation mondiale ? Pour répondre à cette question, on peut faire l'hypothèse que la croissance est linéaire (les quatre points semblent alignés). On cherche alors l'équation de la droite de "meilleure approximation" (dans un sens qu'on ne va pas expliquer ici) de ces quatre points et on utilisera cette expression analytique pour faire cette prévision. Pour cela, nous allons utiliser la fonction `polyfit` du module `numpy` et afficher les points précédent, la courbe d'approximation et le point de prévision :

```
import matplotlib.pyplot as plt
year = [1950, 1970, 1990, 2010]
pop = [2.519, 3.692, 5.263, 6.972]
plt.scatter(year,pop, marker="o", color = "red",
 ↪  label="Data")

from numpy import array, polyfit
# On cherche les coefficients de la droite
# pop = a * year + b
a, b = polyfit(year, pop, 1)
yearN = year + [2030]
plt.plot(yearN, a * array(yearN) + b, 'x--',
 ↪  color="green", label="Prévision")

plt.grid()
plt.xlabel("Années")
plt.ylabel("Population (en milliards)")
plt.legend()
# plt.show()
plt.savefig('Images/populationBIS.png')
```



8.1. Importation des modules matplotlib et numpy

Pour réaliser des graphiques scientifiques, nous pouvons faire appel au module `matplotlib` en l'associant au module `numpy` (dont on a parlé à la page 268). Il existe deux façons d'importer ces modules :

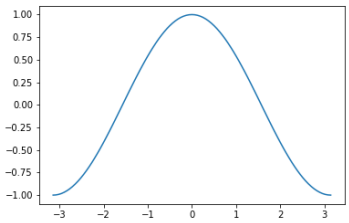
- La façon usuelle consiste à utiliser les instructions suivantes :

```
import matplotlib.pyplot as plt
import numpy as np
```

  Avec cette approche, chaque fonction de `matplotlib` devra être précédée de `plt`, et chaque fonction de `numpy` de `np`, comme dans l'exemple qui suit

```
import matplotlib.pyplot as plt
import numpy as np
xx = np.linspace(-np.pi,np.pi,101)
yy = np.cos(xx)
plt.plot(xx,yy)
plt.show()
```

  On verra plus tard la signification de chaque instruction.



Deux points à noter :

- Il est inutile d'importer le module `math`, car `numpy` redéfinit les mêmes fonctions.
- Les fonctions de `numpy` sont vectorisées, ce qui signifie que si on les applique à une liste **x** (en réalité, un tableau `numpy`), elles génèrent une liste **y** qui contient les évaluations en chaque point de **x**. Dans notre exemple, l'écriture `np.cos(xx)` équivaut à `[np.cos(x) for x in xx]`, car la fonction cosinus utilisée est celle définie par le module `numpy`.

On peut également importer les deux modules simultanément en utilisant le sous-module `matplotlib.pylab`, qui combine les fonctionnalités de `pyplot` (pour les tracés) avec celles de `numpy` (pour la manipulation de tableaux et les opérations mathématiques vectorisées).

- Méthode pour importer les deux modules simultanément :

```
import matplotlib.pylab as pyl
```

De cette manière, on importe également le module `numpy` avec le même alias. Dans ce cas, toutes les instructions (qu'elles viennent de `matplotlib` ou de `numpy`) doivent être précédées de `pyl`. Voici un exemple :

```
import matplotlib.pylab as pyl
xx = pyl.linspace(-pyl.pi,pyl.pi,101)
pyl.plot(xx,pyl.cos(xx))
pyl.show()
```

- Pour les plus "paresseux", il est possible d'utiliser l'instruction

```
from matplotlib.pylab import *
```

Cette méthode importe également les deux modules sans alias. Elle offre un environnement très similaire à celui de MATLAB/Octave, qui est très répandu dans le domaine du calcul scientifique. Voici un exemple :

```
from matplotlib.pylab import *
xx = linspace(-pi,pi,101)
plot(xx,cos(xx))
show()
```

8.2. Tracé d'une courbe sur un repère

Pour tracer le graphe d'une fonction $f : [a,b] \to \mathbb{R}$, il faut tout d'abord générer une liste de points x_i où évaluer la fonction f, puis la liste des valeurs $f(x_i)$ et enfin, avec la fonction `plot`, Python reliera entre eux les points $(x_i, f(x_i))$ par des segments. Plus les points sont nombreux, plus le graphe est proche du graphe de la fonction f.

Pour générer les points x_i on peut utiliser l'une des deux instructions suivantes fournies par le module `numpy` :

- soit l'instruction `np.linspace(a,b,n+1)` qui construit la liste de $n+1$ éléments

$$[a, a+h, a+2h, \dots, b = a+nh] \quad \text{avec } h = \frac{b-a}{n}$$

- soit l'instruction `np.arange(a,b,h)` qui construit la liste de $n = \mathrm{E}\left(\frac{b-a}{h}\right) + 1$ éléments

$$[a, a+h, a+2h, \dots, a+nh]$$

Dans ce cas, attention au dernier terme : avec des `float` les erreurs d'arrondis pourraient faire en sort que b ne soit pas pris en compte.

Canevas

pour l'affichage du graphe de la fonction $f : [a;b] \to \mathbb{R}$ définie par $y = f(x)$ avec $n+1$ points :

```
import matplotlib.pyplot as plt
import numpy as np

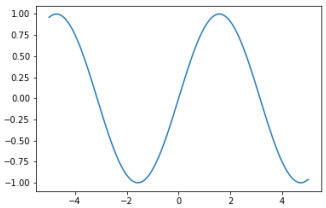
f = lambda x : # à compléter
a,b,n = # à compléter

xx = np.linspace(a,b,n+1) # n+1 points, n sous-intervalles de largeur (b-a)/n
yy = [f(x) for x in xx] # si f est une fonction numpy, on écrira yy = np.f(xx)
plt.plot(xx,yy)
plt.show()
```

Rappelez-vous : la fonction `plot` informe Python sur ce qu'il doit dessiner et comment le dessiner, tandis que `show` affiche réellement le graphique.

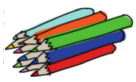
Un exemple avec une sinusoïde :

```
import matplotlib.pyplot as plt
import numpy as np
xx = np.linspace(-5,5,101) # x = [-5,-4.9,-4.8,...,5]
# xx = np.arange(-5,5,0.1) # idem
yy = np.sin(xx)  # fonction vectorisee
plt.plot(xx,yy)
plt.show()
```

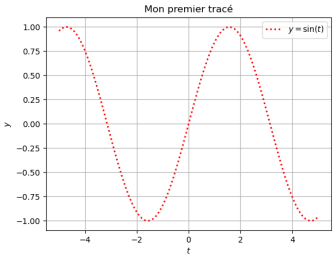


On obtient une courbe sur laquelle on peut zoomer, modifier les marges et sauvegarder dans différents formats.

On peut spécifier la couleur (les abréviations sont chaque lettre de "rgb" et "cymk" avec en plus "w" pour *white*) et le type de trait, changer les étiquettes des axes, donner un titre, ajouter une grille, une légende etc.



```
import matplotlib.pyplot as plt
import numpy as np
xx = np.linspace(-5,5,101)
yy = np.sin(xx)
plt.plot(xx,yy,color='r',ls=":",lw=2,label=r'$y=\sin(t)$')
plt.legend()
plt.grid()
plt.xlabel('$t$')
plt.ylabel('$y$')
plt.title('Mon premier tracé')
plt.show()
```

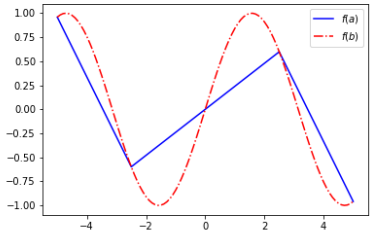


8.3. Plusieurs courbes sur le même repère et options

On peut tracer plusieurs courbes dans le même repère.

Par exemple, dans la figure suivante, on a tracé la même fonction : la courbe bleu correspond à la grille la plus grossière, la courbe rouge correspond à la grille la plus fine :

```
import matplotlib.pyplot as plt
import numpy as np
x1 = np.linspace(-5,5,5)
y1 = np.sin(x1)
plt.plot(x1,y1,'b-',label="5 points")
x2 = np.linspace(-5,5,101)
y2 = np.sin(x2)
plt.plot(x2,y2,'r-.',label="101 points" )
plt.legend()
plt.show()
```



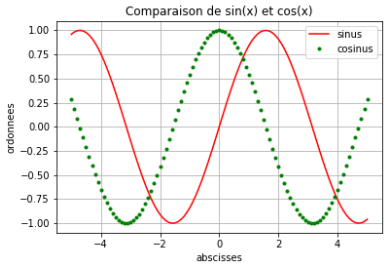
Pour **tracer plusieurs courbes sur le même repère**, on peut les mettre les unes à la suite des autres en spécifiant la couleur et le type de trait, changer les étiquettes des axes, donner un titre, ajouter une grille, une légende etc.

8.3.1. Personnalisation

Quasiment tous les aspects d'une figure peuvent être configurés par l'utilisateur soit pour y ajouter des données, soit pour améliorer l'aspect esthétique. Plutôt que de vous faire une liste des fonctions qui permettent de faire ces actions, voici des exemples.

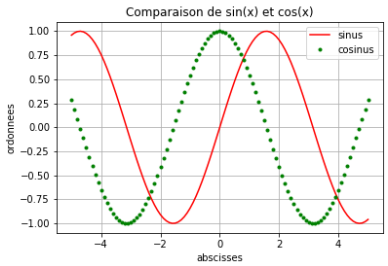
Par exemple, dans le code ci-dessous `"r-"` indique que la première courbe est à tracer en rouge (red) avec un trait continu, et `"g."` que la deuxième est à tracer en vert (green) avec des points.

```
import matplotlib.pyplot as plt
import numpy as np
x  = np.linspace(-5,5,101)
y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x,y1,"r-",x,y2,"g.")
plt.legend(['sinus','cosinus'])
plt.xlabel('abscisses')
plt.ylabel('ordonnees')
plt.title('Comparaison de np.sin(x) et np.cos(x)')
plt.grid(True)
plt.show()
```



soit encore

```
import matplotlib.pyplot as plt
import numpy as np
x  = np.linspace(-5,5,101)
y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x,y1,"r-",label=('sinus'))
plt.plot(x,y2,"g.",label=('cosinus'))
plt.legend()
plt.xlabel('abscisses')
plt.ylabel('ordonnees')
plt.title('Comparaison de np.sin(x) et np.cos(x)')
plt.grid(True)
plt.show()
```



On peut être plus explicite et au lieu de donner un string du type `'r--'` indiquer explicitement qu'il s'agit d'une couleur et d'un style de ligne.

- Les couleurs de base sont
  `{b: blue, g: green, r: red, c: cyan, m: magenta, y: yellow, k: black, w: white}` (les abréviations sont chaque lettre de "rgb" et "cymk" avec en plus "w" pour *white*).
  Pour les utiliser, on écrira `plt.plot(x,y,'r')` ou `plt.plot(x,y,color='red')` ou encore `plt.plot(x,y,c='red')`.
  On peut utiliser aussi un nom HTML comme `plt.plot(x,y,color="darkgreen")`. On peut même utiliser un nom tiré de l'enquête sur les couleurs de xkcd, préfixé par `'xkcd:'` comme
  `plt.plot(x,y,c='xkcd:sky blue')`.
  On peut sinon passer le code HEX de la couleur comme `plt.plot(x,y,color="#f44265")` ou un tuple RGB comme `plt.plot(x,y,color=(0.9569, 0.2588, 0.3891) )`.
- Pour définir le type de ligne, on écrira `plt.plot(x,y,'--')` ou `plt.plot(x,y,linestyle='dashed')` ou encore `plt.plot(x,y,ls='dashed')`. Les 4 styles de lignes de base peuvent être définis à l'aide des chaînes `"solid"`, `"dotted"`, `"dashed"` ou `"dashdot"`, qu'on peut écrire de manière abrégée respectivement comme `"-"`, `":"`, `"--"`, `"-."`. Un contrôle plus fin peut être obtenu en fournissant un tuple de tirets `(offset, (on_off_seq))`. Par exemple, `(0, (3, 10, 1, 15))` signifie "(ligne de 3 pt, espace de 10 pt, ligne de 1 pt, espace de 15 pt)" sans décalage, tandis que `(5, (10, 3))`, signifie "(ligne de 10 pt, espace de 3 pt)", mais en sautant la première ligne de 5 pt.
- Pour définir l'épaisseur de la ligne, on écrira par exemple `plt.plot(x,y,linewidth=2)` ou encore l'abréviation `plt.plot(x,y,lw=2)` (l'unité est le point).
- Pour définir un marker, on écrira `plt.plot(x,y,'o')` ou `plt.plot(x,y,marker='o')`. Dans le tableau on a indiqué une partie des marker disponibles. Pour la liste complète on écrira

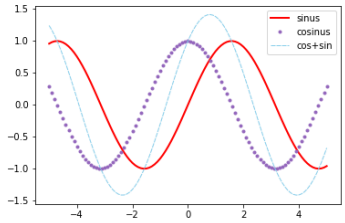
  ```
  from matplotlib.lines import Line2D
  print(Line2D.markers, sep=", ", end="")
  ```

  ```
  {'.': 'point', ',': 'pixel', 'o': 'circle', 'v': 'triangle_down', '^': 'triangle_up', '<':
  ↪  'triangle_left', '>': 'triangle_right', '1': 'tri_down', '2': 'tri_up', '3': 'tri_left', '4':
  ↪  'tri_right', '8': 'octagon', 's': 'square', 'p': 'pentagon', '*': 'star', 'h': 'hexagon1', 'H':
  ↪  'hexagon2', '+': 'plus', 'x': 'x', 'D': 'diamond', 'd': 'thin_diamond', '|': 'vline', '_':
  ↪  'hline', 'P': 'plus_filled', 'X': 'x_filled', 0: 'tickleft', 1: 'tickright', 2: 'tickup', 3:
  ↪  'tickdown', 4: 'caretleft', 5: 'caretright', 6: 'caretup', 7: 'caretdown', 8: 'caretleftbase',
  ↪  9: 'caretrightbase', 10: 'caretupbase', 11: 'caretdownbase', 'None': 'nothing', 'none':
  ↪  'nothing', ' ': 'nothing', '': 'nothing'}
  ```

  Il est possible d'en fixer la taille par les commandes `plt.plot(x,y,markersize=5)` ou encore `plt.plot(x,y,ms=5)` (l'unité est le point).

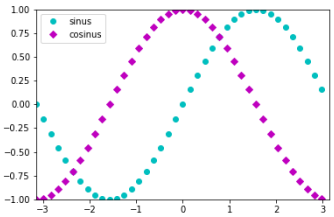
Voici un exemple d'utilisation :

```
import matplotlib.pyplot as plt
import numpy as np
x  = np.linspace(-5,5,101)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.sin(x)+np.cos(x)
plt.plot(x,y1,color='r'       ,ls='-'       ,linewidth=2            ,label='sinus')
plt.plot(x,y2,color='purple' ,ls=' '       ,lw=1        ,marker='.',label='cosinus')
plt.plot(x,y3,color='skyblue',ls=(0,(5,1)),lw=1                    ,label='cos+sin')
plt.legend()
plt.show()
```



On peut déplacer la légende en spécifiant l'une des valeurs suivantes : `best`, `upper right`, `upper left`, `lower right`, `lower left`, `center right`, `center left`, `lower center`, `upper center`, `center` :

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(-np.pi,np.pi,0.05*np.pi)
plt.plot(x,np.sin(x),'co', x,np.cos(x),'mD')
plt.legend(['sinus','cosinus'],loc='upper left')
# plt.axis([xmin, xmax, ymin, ymax])
plt.axis([-np.pi, pi, -1, 1]);
plt.show()
```



On peut personnaliser l'aspect des étiquettes des axes, par exemple

```
plt.xlabel("Time (s)", size = 16, family='serif', color='r', weight='normal', labelpad = 6)
```

- Pour `size`, il s'agit de la taille (en points).
- Pour `family`, on peut choisir parmi `'serif'`, `'sans-serif'`, `'cursive'`, `'fantasy'`, `'monospace'`
- Pour la couleur, on a les mêmes possibilités que pour les lignes.
- Pour `weight`, on peut choisir parmi `'light'`, `'normal'`, `'medium'`, `'semibold'`, `'bold'`, `'heavy'` et `'black'`.
- Le `labelpad` est la distance entre l'axe et l'étiquette.

On peut personnaliser le titre du repère, par exemple

```
plt.title("Title Example", loc='right', fontdict={'family': 'serif', 'color' : 'darkblue', 'weight':
↪  'bold', 'size': 18})
```

La liste complète des options est accessible en utilisant la fonction `help` :

```
import matplotlib.pyplot as plt
print( help(plt.plot) )
```

Voir aussi

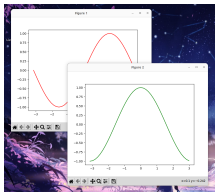
- https://matplotlib.org/api/markers_api.html
- https://matplotlib.org/examples/lines_bars_and_markers/marker_reference.html
- https://matplotlib.org/gallery/color/named_colors.html#sphx-glr-gallery-color-named-colors-py
- https://matplotlib.org/examples/lines_bars_and_markers/linestyles.html
- https://www.delftstack.com/tutorial/matplotlib/

8.4. Plusieurs repères dans des fenêtres distinctes

Afficher plusieurs graphiques dans des fenêtres distinctes est une approche courante pour visualiser différentes informations graphiques simultanément. Cette technique permet de créer des fenêtres séparées pour chaque graphique, offrant ainsi une vue détaillée et ciblée sur chaque visualisation.

Avec `plt.figure()` on génère une nouvelle fenêtre. Chaque fenêtre de graphique peut être personnalisée individuellement avec des axes, des titres, des étiquettes et des styles spécifiques.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(-np.pi,np.pi,0.05*np.pi)
plt.figure(1)
plt.plot(x, np.sin(x), 'r')
plt.figure(2)
plt.plot(x, np.cos(x), 'g')
plt.show()
```



Pour modifier la taille de la figure on utilisera `figsize`, comme par exemple `plt.figure(1,figsize=(3,3))`.

8.5. Plusieurs repères dans une grille au sein d'une même fenêtre

La fonction `fig,ax=plt.subplots(r,c)` subdivise la fenêtre (appelée `fig`) en une grille (appelée `ax`) de `r` lignes et `c` colonnes. Par exemple, si on écrit `plt.subplots(4,3)`, cela signifie que la fenêtre principale a été subdivisée en $4 \times 3 = 12$ cases.

8.5.1. Approche I

Chaque case est numérotée avec un seul indice si la grille contient une seule ligne ($r = 1$) et avec deux indices si $r > 1$. Pour pouvoir utiliser un seul indice dans tous les cas, nous utiliserons la commande `reshape`.

On peut alors parcourir les indices de la matrice comme un simple vecteur : les cases sont numérotées de gauche à droite, du haut vers le bas, à partir de l'indice 0. Par exemple, la case qui dans la matrice correspond à la ligne d'indice 0 et à la colonne d'indice 2 (celle en haut à droite) sera numéroté 2.

① Exemple avec une grille qui contient une seule ligne de repères (notons qu'on peut afficher plusieurs courbes sur le même repère) :

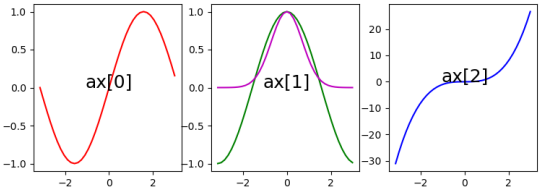
```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(-np.pi, np.pi, 0.05*np.pi)

fig,ax = plt.subplots(1,3,figsize=(9,3))

ax[0].plot(x, np.sin(x), 'r')
ax[1].plot(x, np.cos(x), 'g')
ax[1].plot(x, np.exp(-x*x), 'm')
ax[2].plot(x, x**3, 'b')

plt.show()
```



② Exemple avec une grille de repères :

```
import matplotlib.pyplot as plt
import numpy as np

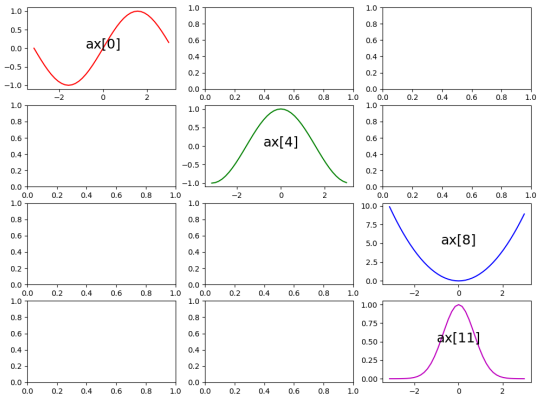
x = np.arange(-np.pi, np.pi, 0.05*np.pi)

fig,ax = plt.subplots(4,3,figsize=(12,9))
ax = ax.reshape(-1)

ax[0].plot(x, np.sin(x), 'r')
ax[4].plot(x, np.cos(x), 'g')
ax[8].plot(x, x*x, 'b')
ax[11].plot(x, np.exp(-x*x), 'm')

# On peut effacer les repères inutilisés:
# for i in [1,2,3,5,6,7,9,10]:
#     fig.delaxes(ax[i])

plt.show()
```



8.5.2. Approche II (à la MATLAB)

Une autre approche est la suivante : la fonction `plt.subplot(r,c,i)` (sans "**s**") subdivise la fenêtre sous forme d'une matrice de `r` lignes et `c` colonnes où chaque case est numérotée et `i` est le numéro de la case où afficher le graphe. La numérotation se fait de gauche à droite, puis de haut en bas, en commençant par 1 (bien noter la différence avec l'approche précédente où la numérotation commençait par 0). Par exemple, si on écrit `plt.subplot(4,3,3)`, cela signifie que la fenêtre principale a été subdivisée en $4 \times 3 = 12$ cases ; la case qui a pour indice 3 est celle en haut à droite.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(-np.pi, np.pi, 0.05*np.pi)

r, c = 4, 3

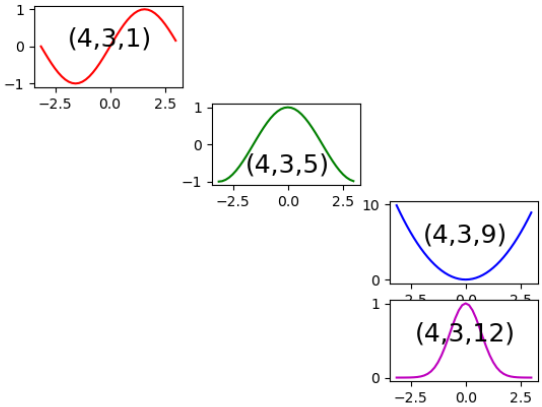
plt.subplot(r, c, 1)
plt.plot(x, np.sin(x), 'r')

plt.subplot(r, c, 5)
plt.plot(x, np.cos(x), 'g')

plt.subplot(r, c, 9)
plt.plot(x, x*x, 'b')

plt.subplot(r, c, 12)
plt.plot(x, np.exp(-x*x), 'm')

plt.show()
```



8.6. ☆ Animations

`FuncAnimation` est une fonction de la classe `matplotlib.animation`. Elle permet de créer une animation à partir d'une fonction.

Nous allons construire un exemple simple d'animation d'un point le long d'une fonction sinusoïdale. On commence par importer les modules nécessaires :

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
```

Nous allons ensuite tracer la fonction sinus :

```
my_fig = plt.figure()
xx = np.arange(0, 2*np.pi, 0.001)
yy = np.sin(xx)
plt.plot(xx,yy)
```

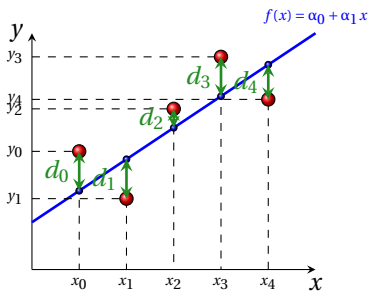
Ensuite, nous allons créer le point rouge que nous allons ensuite animer. Pour créer le point rouge, nous commençons par $(0, \sin(0))$. Nous utilisons `dot,` car `plt.plot` renvoie un tuple (et nous ajoutons la virgule après la variable pour décompresser le tuple).

```
dot, = plt.plot([0], [np.sin(0)], 'ro') # NB la virgule après dot!!!
def my_func(i):
    dot.set_data(i, np.sin(i))
    return dot, # NB la virgule !!!

_animation = FuncAnimation(fig=my_fig, func=my_func, frames=np.arange(0, 2*np.pi, 0.1), interval=10)
plt.show()
```

8.7. ☆ Régression : polynôme de meilleure approximation

Supposons que deux grandeurs x et y sont liées approximativement par une relation affine, *i.e.* y est peu différent de $f(x) = \alpha_0 + \alpha_1 x$ (autrement dit, lorsqu'on affiche ces points dans un plan cartésien, les points ne sont pas exactement alignés mais cela semble être dû à des erreurs de mesure). On souhaite alors trouver les constantes α_0 et α_1 pour que la droite d'équation $y = \alpha_0 + \alpha_1 x$ s'ajuste *le mieux possible* aux points observés.



Pour cela, introduisons $d_i(\alpha_0, \alpha_1) \equiv y_i - (\alpha_0 + \alpha_1 x)$ l'écart

vertical du point (x_i, y_i) par rapport à la droite.
La méthode des moindres carrés est celle qui choisit α_0 et α_1 de sorte que *la somme des carrés de ces écarts soit minimale.*
La droite d'équation $y = \alpha_1 x + \alpha_0$ ainsi calculée s'appelle *droite de régression de y par rapport à x.*

On peut généraliser cette approche en supposant que les deux grandeurs x et y sont liées par une relation polynomiale, c'est-à-dire de la forme $y = \sum_{j=0}^{m} a_j x^j$ pour certaines valeurs de a_j. On souhaite alors trouver les $m+1$ constantes a_j pour que le polynôme d'équation $f(x) = \sum_{j=0}^{m} a_j x^j$ s'ajuste le mieux possible aux points observés.

La fonction au cœur de la régression est `polyfit` du module `numpy`. Elle s'utilise de la façon suivante :

```
import numpy as np
coeff = np.polyfit(xx,yy,n)
```

où `xx` et `yy` désignent respectivement la liste des abscisses et des ordonnées des points du nuage de points et n est le degré du polynôme de meilleure approximation. `coeff` est un tuple qui contient les coefficients du polynôme cherché, dans l'ordre décroissant des puissances. Par exemple, si on veut approcher le nuage de points par un polynôme de la forme $ax^2 + bx + c$, on écrira `a,b,c = np.polyfit(xx,yy,2)`.

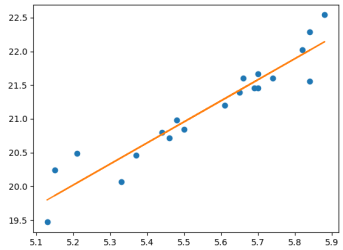
◇

Exemple de régression linéaire :

```
import matplotlib.pyplot as plt
import numpy as np

# Les listes des abscisses et ordonnées :
xx = [5.13, 5.7, 5.48, 5.7, 5.66, 5.84, 5.5, 5.69,
↪  5.44, 5.84, 5.82, 5.88, 5.61, 5.65, 5.21, 5.37,
↪  5.46, 5.33, 5.15, 5.74]
yy = [19.47, 21.67, 20.98, 21.46, 21.6, 22.29,
↪  20.84, 21.46, 20.8, 21.56, 22.02, 22.54, 21.2,
↪  21.39, 20.49, 20.46, 20.72, 20.07, 20.24, 21.6]

a,b = np.polyfit(xx,yy,1)
plt.plot(xx, yy, "o")
plt.plot(xx, a*xx+b)
plt.show()
```



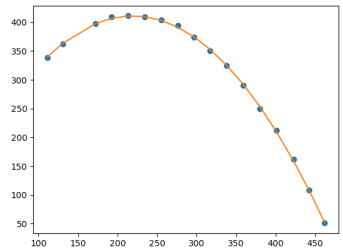
◇

Exemple de régression d'ordre 2 :

```
import matplotlib.pyplot as plt
import numpy as np

xx = [111, 131, 172, 192, 213, 234, 255, 276, 297,
↪  317, 338, 359, 380, 401, 423, 442]
yy = [339, 362, 398, 409, 411, 409, 404, 394, 374,
↪  350, 325, 290, 250, 212, 162, 108]

a,b,c = np.polyfit(xx,yy,2)
plt.plot(xx, yy, "o")
plt.plot(xx, [a*x**2+b*x+c for x in xx])
plt.show()
```



Même si la relation entre deux quantités n'est pas linéaire, il est parfois possible d'appliquer une transformation pour trouver une relation linéaire.

***Fitting* linéaire après transformation d'un exponentiel** Soit $a > 0$ et considérons la fonction $f(x) = ae^{kx}$: elle est non-linéaire mais si on prend son logarithme on obtient $\ln(f(x)) = \ln(a) + kx$ qui est linéaire et a la forme $\alpha_0 + \alpha_1 x$ avec $\alpha_1 = k$ et $\alpha_0 = \ln(a)$.
On peut alors calculer l'équation de la droite de régression sur l'ensemble $\{(x_i, \ln(y_i))\}_{i=0}^{n}$ et obtenir ainsi k et $\ln(a)$.

***Fitting* linéaire après transformation d'une puissance** Soit $a > 0$ et considérons la fonction $f(x) = ax^k$: elle est non-linéaire mais si on prend son logarithme on obtient $\ln(f(x)) = \ln(a) + k\ln(x)$ qui est linéaire et a la forme $\alpha_0 + \alpha_1 x$ avec $\alpha_1 = k$ et $\alpha_0 = \ln(a)$.
On peut alors calculer l'équation de la droite de régression sur l'ensemble $\{(\ln(x_i), \ln(y_i))\}_{i=0}^{n}$ et obtenir ainsi k et $\ln(a)$.

EXEMPLE (ÉTUDE D'ORDRE DE CONVERGENCE.)
Généralement, un processus numérique est une approximation d'un modèle mathématique obtenue en fonction d'un paramètre de discrétisation h, supposé positif. On espère que lorsque $h \to 0$, le processus numérique converge vers la solution du modèle mathématique. Dans ce cas, on dit que le processus numérique est **convergent**. Si l'erreur (absolue ou relative) peut être bornée par e_c une fonction de h telle que $e_c \leq \mathrm{C}h^p$ où C est une constante indépendante de h, et p est généralement un entier, on dit que la **méthode est convergente d'ordre** p. Dans les études de convergence, il est courant d'analyser des graphiques où l'erreur est tracée en fonction de h sur une **échelle logarithmique**. Sur ces graphiques l'axe des abscisses représente $\log(h)$, l'axe des ordonnées représente $\log(e_c)$. L'avantage est que si $e_c \simeq \mathrm{C}h^p$, alors

$$\log(e_c) \simeq \log(\mathrm{C}) + p\log(h).$$

Ainsi, en échelle logarithmique, p correspond à la pente de la droite $\log(e_c)$. Si l'on compare deux méthodes, celle ayant la pente la plus forte sera d'ordre plus élevé.

Pour calculer les coefficients de la droite de régression linéaire dans un graphique logarithmique, on peut utiliser la méthode `polyfit` de la bibliothèque `numpy`, qui permet de trouver les coefficients p et b d'une droite de meilleure approximation en utilisant la méthode des moindres carrés.

EXEMPLE (ÉTUDE DE COMPLEXITÉ.)
Considérons un algorithme qui dépend d'un entier n. On veut savoir comment le temps t d'exécution évolue lorsqu'on n augmente. Par exemple, si la croissance est polynomiale, on cherchera à calculer p tel que $n \mapsto t(n) \approx cn^p$. Pour cela on calcule d'abord le temps d'exécution pour différents valeurs de n, puis on affiche $n \mapsto t(n)$ en échelle logarithmique. On obtient une droite ($\ln(t) = \ln(c) + p\ln(n)$) de pente p. Pour estimer cette pente on utilisera `polyfit`.

Dans l'exemple suivant on estime la complexité de l'ajout d'un terme dans une liste par la méthode `append`. Pour chaque valeur de N, on calcule combien de temps est nécessaire pour remplir la liste et on le sauvegarde. À la fin on aura un ensemble de points $\{(\mathrm{N}_i, \mathrm{T}_i)\}_{i=0}^4$ et on pourra chercher la droite de meilleur approximation de l'ensemble $\{(\ln(\mathrm{N}_i), \ln(\mathrm{T}_i)\}_{i=0}^4$.

```
import numpy as np
from time import perf_counter

NN = [ 10**2, 10**3, 3*10**3, 5*10**3 ]
TT = []

for N in NN:
    debut=perf_counter()
    L = []
    for i in range(N):
        for j in range(N):
            L.append(i+j)
    fin = perf_counter()
    TT.append(fin-debut)

p,b = np.polyfit(np.log(NN),np.log(TT),1)
print(f"Pente = {p}")

# # AFFICHAGE
# import matplotlib.pyplot as plt
# plt.loglog(NN,TT,"o")
# plt.loglog(NN,[np.exp(b)*n**p for n in NN])
# plt.show()
```

8.8. Exercices

Canevas

pour l'affichage du graphe de la fonction $f\colon [a;b] \to \mathbb{R}$ définie par $y = f(x)$ avec $n+1$ points :

```
import matplotlib.pyplot as plt
import numpy as np

f = lambda x : # à compléter
a, b, n = # à compléter

xx = np.linspace(a,b,n+1) # n+1 points, n sous-intervalles de largeur (b-a)/n
yy = [f(x) for x in xx] # pour une fonction numpy, on écrira yy = np.f(xx)

plt.plot(xx,yy)
plt.show()
```

Exercice 8.1 (Le gateau)

Une tarte est partagée entre 100 invités. Le premier invité reçoit 1% de la tarte, le deuxième 2% du reste, ... le i-ème invité reçoit i% du reste. Le dernier invité reçoit donc 100% du reste, ce qui consomme la tarte intégralement. Qui reçoit la plus grande part de tarte ? Pour résoudre ce problème, commencer par calculer la part de tarte reçue par chaque invité et afficher un graphique avec les invités en abscisses et le pourcentage de tarte qu'ils reçoivent en ordonnées. Identifier ensuite celui qui reçoit la plus grande part.

Correction

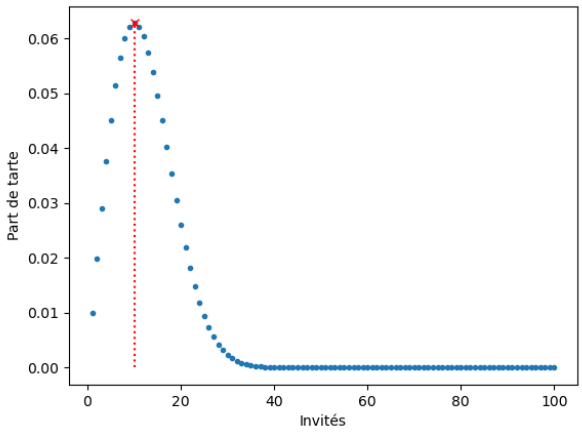
```
import matplotlib.pyplot as plt

L, tot, invites = [], 1, 100
for i in range(1, invites+1):
    L.append(i / invites * tot)
    tot -= i / invites * tot

max_val = max(L)
max_idx = L.index(max_val)
print(f"Le {max_idx + 1}-ème invité reçoit la part
↪  la plus grande, soit {max_val * 100:.2f}% de la
↪  tarte.")

plt.plot(range(1,invites+1), L, '.')
plt.plot(max_idx+1, max_val, 'rx')
plt.plot([max_idx+1, max_idx+1], [0, max_val],
↪  'r:')
plt.xlabel('Invités')
plt.ylabel('Part de tarte')
# plt.savefig("Images/gateau.png")
plt.show()
```

```
Le 10-ème invité reçoit la part la plus grande,
↪  soit 6.28% de la tarte.
```



Exercice 8.2 (Bhaskara I)

Tracer dans le même repère le graphe représentatif des fonctions

$$f(x) = \sin(x), \qquad g(x) = \frac{16(\pi - x)x}{5\pi^2 - 4(\pi - x)x}, \qquad x \in [0, \pi].$$

Ajouter une grille, la légende, un titre.

Même exercice pour

$$f(x) = \cos(x), \qquad g(x) = \frac{\pi^2 - 4x^2}{\pi^2 + x^2}, \qquad x \in [0, \pi].$$

Correction
Bhāskara I est un mathématicien indien du VII siècle. Il donna cette unique et remarquable approximation rationnelle de la fonction sinus.

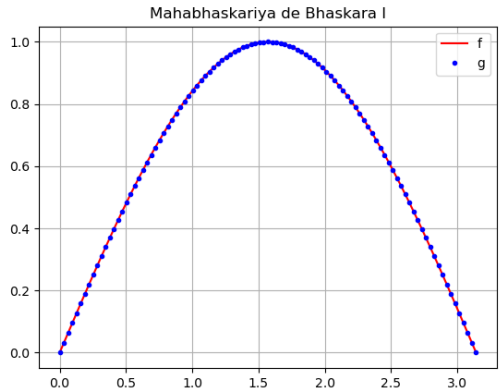
Source : https://scholarworks.umt.edu/cgi/viewcontent.cgi?article=1313&context=tme

```
import matplotlib.pyplot as plt
import numpy as np

f = lambda x : np.sin(x)
g = lambda x :
↪  16*x*(np.pi-x)/(5*np.pi**2-4*(np.pi-x)*x)

xx = np.linspace(0,np.pi,101)
yy = [f(x) for x in xx]
zz = [g(x) for x in xx]

plt.plot(xx,yy,'r-',label=('f'))
plt.plot(xx,zz,'b.',label=('g'))
plt.title("Mahabhaskariya de Bhaskara I")
plt.grid()
plt.legend(loc="best")
plt.show()
```



Notons qu'on peut même écrire `yy = f(xx)` et `zz = g(xx)` car `xx` est un vecteur numpy (les opérations élémentaires sont vectorisées).

Exercice 8.3 (Tracer des droites)
Tracer sur $[-2;2]$ dans un même repère les droites d'équation $f(x) = x+1$, $g(x) = 2.5$ (droite horizontale) et $x = 1.5$ (droite verticale). Ajouter une grille, la légende, un titre.

Correction

```
import matplotlib.pyplot as plt
import numpy as np

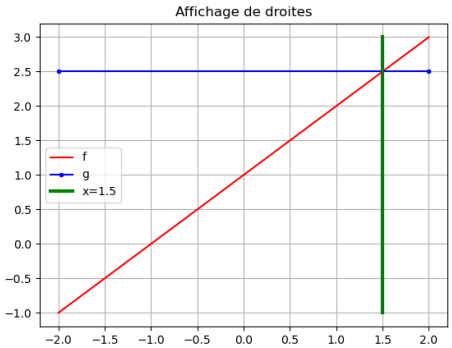
xx = np.linspace(-2,2,101)

f  = lambda x : x+1
ff = [f(x) for x in xx]
plt.plot(xx,ff,'r-',label=('f'))

# horizontale
g  = lambda x : 2.5
gg = [g(x) for x in xx]
plt.plot(xx,gg,'r-',label=('g'))

# verticale
yy = np.linspace(-2,2,101)
h  = lambda y : 1.5
hh = [h(y) for y in yy]
plt.plot(hh,yy,'g-',lw=3,label=('x=1.5'))

plt.title("Affichage de droites")
plt.grid()
plt.legend(loc="best")
plt.show()
```



Notons qu'on peut même écrire `yy = xx+1` car `xx` est un vecteur numpy (les opérations élémentaires sont vectorisées).

Étant donné que pour définir un segment il suffit d'imposer le passage par deux points, on peut tracer les mêmes graphes comme ci-dessous :

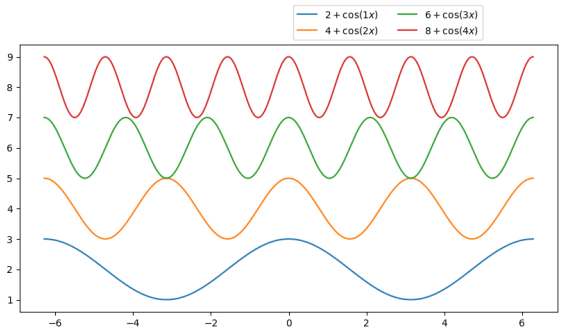
```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([-2,2],[-1,3],'r-',label=('f')) # segment d'extrémités (-2,-1) et (2,3)
```

```
plt.plot([-2,2],[2.5,2.5],'b.-',label=('g')) # segment d'extrémités (-2,2.5) et (2,2.5)
plt.plot([1.5,1.5],[-1,3],'g-',lw=3,label=('x=1.5')) # segment d'extrémités (1.5,-1) et (1.5,3)
plt.show()
```

Exercice 8.4 (Tracer plusieurs courbes)

Tracer dans le même repère le graphe des fonctions $f_n(x) = 2n + \cos(nx)$ pour $n = 1,2,3,4$.

Correction



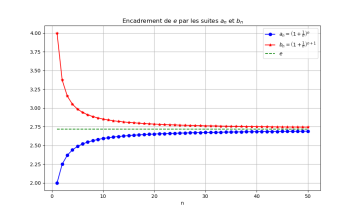

```
import matplotlib.pyplot as plt
import numpy as np
plt.figure(num=None, figsize=(10, 5))
f = lambda x,n : 2*n+np.cos(n*x)
xx = np.linspace(-2*np.pi,2*np.pi,1001)
for n in range(1,5):
    yy = [f(x,n) for x in xx]
    plt.plot(xx,yy, label=fr'${2*n}+\cos({n}x)$')
plt.legend(bbox_to_anchor=(0.5, 1),ncol=2);
plt.show()
```

Exercice 8.5 (Encadrement de e)

Soient $(a_n)_{n\in\mathbb{N}^*}$ et $(b_n)_{n\in\mathbb{N}^*}$ deux suites définies par

$$a_n = \left(1+\frac{1}{n}\right)^n, \qquad b_n = \left(1+\frac{1}{n}\right)^{n+1}.$$

On sait que $a_n < e < b_n$ et que $\lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n = e$. Afficher dans un même repère les deux suites en fonction de n ainsi que la droite horizontale e.



Correction

```
import numpy as np
import matplotlib.pyplot as plt

# Valeurs de n à tester
nn = np.linspace(1, 50, 50)

# Calcul des valeurs de a_n et b_n
# nn est un vecteur numpy : opérations vectorisées
aa = (1 + 1/nn)**nn
bb = (1 + 1/nn)**(nn+1)

# Tracé des suites a_n et b_n ainsi que la constante e
plt.figure(figsize=(10, 6))
plt.plot(nn, aa, 'bo-', label=r'$a_n = \left(1+\frac{1}{n}\right)^n$')
plt.plot(nn, bb, 'r*-', label=r'$b_n = \left(1+\frac{1}{n}\right)^{n+1}$')
plt.plot([nn[0], nn[-1]], [np.e, np.e], 'g--', label='$e$') # segment

# Configuration du graphique
plt.xlabel('n')
plt.title("Encadrement de $e$ par les suites $(a_n)$ et $(b_n)$")
plt.legend()
plt.grid(True)
plt.show()
```

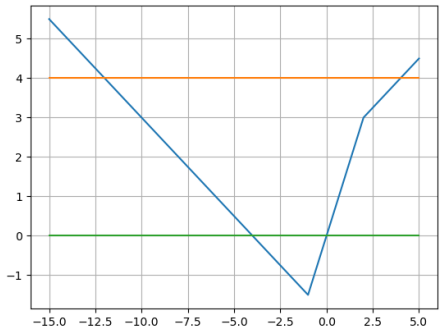
Exercice 8.6 (Tracer une fonction définie par morceaux)

Tracer le graphe de la fonction $f : \mathbb{R} \to \mathbb{R}$ définie par $f(x) = |x+1| - \left|1 - \frac{x}{2}\right|$. On se propose maintenant de résoudre l'équation $f(x) = 0$ et l'inéquation $f(x) \leq 4$. Trouver d'abord une solution approchée graphiquement puis en utilisant la fonction `fsolve` du module `scipy.optimize` dont la syntaxe est `fsolve(f,x0)` (x_0 est un point pas trop éloigné de la solution cherchée). Cette fonction renvoie une liste contenant la solution (approchée) et une estimation de l'erreur.

Bonus : même question avec le module pour le calcul formel `sympy`.

Correction

En affichant une grille et en utilisant le zoom, on peut conjecturer que $f(x) = 0$ pour $x \approx -4$ et $x \approx 0$. On affiche alors $f(-4)$ et $f(0)$. De même, on peut conjecturer que $f(x) \leq 4$ pour $-12 \lesssim x \lesssim 4$. On affiche alors $f(-12)$ et $f(4)$.




```
import matplotlib.pyplot as plt
import numpy as np
f  = lambda x : abs(x+1)-abs(1-x/2)
xx = np.linspace(-15,5,101)
yy = [f(x) for x in xx]
plt.plot(xx,yy)
plt.plot([-15,5],[4,4])
plt.plot([-15,5],[0,0])
plt.grid()
plt.show()
print(f"f(-4)={f(-4)}, f(0)={f(0)}")
print(f"f(-12)={f(-12)}, f(4)={f(4)}")
```

```
f(-4)=0.0, f(0)=0.0 f(-12)=4.0, f(4)=4.0
```

Variante : il s'agit d'une fonction affine par morceaux, on peut donc juste relier des segments :

```
import matplotlib.pyplot as plt
import numpy as np
f  = lambda x : abs(x+1)-abs(1-x/2)
xx = [-15,-1,2,5]
yy = [f(x) for x in xx]
plt.plot(xx,yy)
plt.plot([-15,5],[4,4])
plt.plot([-15,5],[0,0])
plt.grid()
plt.show()
print(f"f(-4)={f(-4)}, f(0)={f(0)}")
print(f"f(-12)={f(-12)}, f(4)={f(4)}")
```

Avec scipy :

```
from scipy.optimize import fsolve
f  = lambda x : abs(x+1)-abs(1-x/2)
print(fsolve(f,-5),fsolve(f,1))
```

```
[-4.] [7.40148683e-17]
```

Bonus :

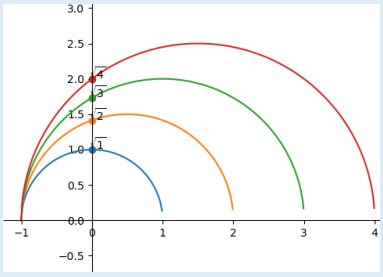
```
import sympy
x = sympy.Symbol('x', real=True)
sol = sympy.solve(abs(x+1)-abs(1-x/2),x)
print(f"f(x)=0 ssi x appartient à l'ensemble {sol}")
print("(NB ce n'est pas un intervalle mais l'ensemble des solutions de l'équation)")
sol = sympy.solve(abs(x+1)-abs(1-x/2)<4,x)
print(f"f(x)<0 ssi {sol}")
```

```
f(x)=0 ssi x appartient à l'ensemble [-4, 0]
(NB ce n'est pas un intervalle mais l'ensemble des solutions de l'équation)
f(x)<0 ssi (-12 < x) & (x < 4)
```

Exercice 8.7 (Représentation graphique des racines carrées)

Considérons un cercle dont un diamètre est le segment qui relie les points $(-1,0)$ et $(k,0)$ pour $k>0$.

1. Écrire l'équation du demi-cercle supérieur en fonction de x et k.
2. Calculer l'intersection de ce demi-cercle avec l'axe des ordonnées et vérifier que cette intersection correspond à $\sqrt{k}$.
3. Pour visualiser cette construction, tracer les arcs de cercle pour $k=1,2,\ldots$ et vérifier que l'intersection avec l'axe des ordonnées est $\sqrt{k}$.



Correction

Le rayon du cercle est donné par $r=\frac{k+1}{2}$ et le centre du cercle est positionné à $x_c=\frac{k-1}{2}$ sur l'axe horizontal (*i.e.*, $y_c=0$). L'équation du cercle est $(x-x_c)^2+(y-y_c)^2=r^2$, ainsi l'équation du demi-cercle supérieur est $f(x,k)=\sqrt{r^2-(x-x_c)^2}$.

Pour trouver l'intersection avec l'axe des ordonnées, nous évaluons $f(x,k)$ en $x=0$:

$$f(0,k)=\sqrt{r^2-x_c^2}=\sqrt{\left(\frac{k+1}{2}\right)^2-\left(\frac{k-1}{2}\right)^2}=\sqrt{\frac{(k+1)^2}{4}-\frac{(k-1)^2}{4}}=\sqrt{\frac{(k^2+2k+1)-(k^2-2k+1)}{4}}=\sqrt{\frac{4k}{4}}=\sqrt{k}.$$

Ainsi, l'intersection avec l'axe des ordonnées est bien $\sqrt{k}$.

```python
import matplotlib.pyplot as plt
import numpy as np

r = lambda k: (k + 1) / 2  # Rayon du cercle
xc = lambda k: (k - 1) / 2  # Centre du cercle sur l'axe x
f = lambda x, k: np.sqrt(r(k)**2 - (x - xc(k))**2)  # Équation de l'arc de cercle

fig, ax = plt.subplots()
ax.axis('equal')

# Tracé des arcs de cercle
xx = np.linspace(-1, 10, 1001)
for k in range(1, 5):
    print(f"Pour {k=}, l'intersection vaut {f(0,k)=}.")
    ax.plot(xx, f(xx, k))
    ax.scatter(0, f(0, k))
    ax.annotate(rf'$\sqrt{{{k}}}$', (0, f(0, k)),
                xytext=(5, 0), ha='center', textcoords='offset points')

# Configuration des axes (bonus)
ax.spines['left'].set_position('zero')   # Positionner l'axe y sur zéro
ax.spines['right'].set_color('none')     # Masquer les graduations de l'axe y droit
ax.spines['bottom'].set_position('zero') # Positionner l'axe x sur zéro
ax.spines['top'].set_color('none')       # Masquer les graduations de l'axe x supérieur

plt.show()
```

```
Pour k=1, l'intersection vaut f(0,k)=1.0.
Pour k=2, l'intersection vaut f(0,k)=1.4142135623730951.
Pour k=3, l'intersection vaut f(0,k)=1.7320508075688772.
Pour k=4, l'intersection vaut f(0,k)=2.0.
```

Exercice 8.8 (Construction alternée des racines carrées)

Soient C_0 et C_1 les points de coordonnées respectives $(0,0)$ et $(0,1)$. Soit u_0 le point de coordonnées $(0,1)$. On construit une suite de points u_n selon la règle suivante :

- pour n pair, on définit le point u_{n+1} comme l'intersection de l'axe $y=0$ avec le cercle de centre C_0 passant par u_n ;

- pour n impair, on définit le point u_{n+1} comme l'intersection de l'axe $y = 1$ avec le cercle de centre C_1 passant par u_n.

1. Montrer par récurrence que la coordonnée x du point u_n est $\sqrt{n}$.
2. Justifier que le rayon utilisé pour passer de u_n à u_{n+1} est $r_n = \sqrt{n+1}$.
3. Implémenter cette construction pour visualiser les premières racines.

Correction

Démonstration par récurrence: soit H_n l'hypothèse $u_n = (\sqrt{n}, y_n)$ avec $y_n = 1$ si n est pair et $y_n = 0$ si n est impair.

- **Initialisation:** Pour $n = 0$, $u_0 = (0,1)$. On a bien $\sqrt{0} = 0$ et l'ordonnée est 1. H_0 est vraie.
- **Hérédité:** Supposons H_n vraie pour un entier $n \geq 0$.
  - **Si n est pair:** $u_n = (\sqrt{n}, 1)$. Le pivot est $C_0(0,0)$. Le rayon est $r_n = \text{dist}(C_0, u_n) = \sqrt{(\sqrt{n}-0)^2 + (1-0)^2} = \sqrt{n+1}$. Le point u_{n+1} est sur l'axe $y = 0$ avec une abscisse positive égale au rayon, donc $u_{n+1} = (\sqrt{n+1}, 0)$.
  - **Si n est impair:** $u_n = (\sqrt{n}, 0)$. Le pivot est $C_1(0,1)$. Le rayon est $r_n = \text{dist}(C_1, u_n) = \sqrt{(\sqrt{n}-0)^2 + (0-1)^2} = \sqrt{n+1}$. Le point u_{n+1} est sur l'axe $y = 1$ avec une abscisse positive égale au rayon, donc $u_{n+1} = (\sqrt{n+1}, 1)$.

  Dans les deux cas, l'abscisse de u_{n+1} est bien $\sqrt{n+1}$, ce qui prouve l'hérédité.

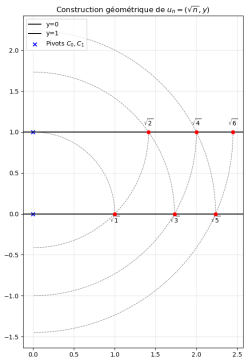
Tableau récapitulatif des étapes:

n	u_n (départ)	Pivot C	Rayon r_n	u_{n+1} (arrivée)
0	$(0,1)$	$C_0(0,0)$	$\sqrt{1}$	$(\sqrt{1},0)$
1	$(\sqrt{1},0)$	$C_1(0,1)$	$\sqrt{2}$	$(\sqrt{2},1)$
2	$(\sqrt{2},1)$	$C_0(0,0)$	$\sqrt{3}$	$(\sqrt{3},0)$
3	$(\sqrt{3},0)$	$C_1(0,1)$	$\sqrt{4}$	$(\sqrt{4},1)$
n	$(\sqrt{n}, y_n)$	$(0, 1-y_n)$	$\sqrt{n+1}$	$(\sqrt{n+1}, 1-y_n)$

```python
import matplotlib.pyplot as plt
import numpy as np
from tabulate import tabulate

# Paramètres de la construction
n_max = 6  # Nombre d'étapes à afficher
u_coords = [(0, 1)]  # u0
pivots = []
rayons = []

# Liste pour stocker les lignes du tableau
table_data = []

fig, ax = plt.subplots(figsize=(10, 4))
ax.set_aspect('equal')

for n in range(n_max):
    # 1. Déterminer le point de départ u_n
    ux_n, uy_n = u_coords[-1]

    # 2. Déterminer le pivot C et la cible
    # Si uy_n = 1 (en haut), on pivote autour de C0(0,0) pour aller vers y=0
    # Si uy_n = 0 (en bas), on pivote autour de C1(0,1) pour aller vers y=1
    if uy_n == 1:
        cx, cy = 0, 0
        target_y = 0
    else:
        cx, cy = 0, 1
        target_y = 1

    # 3. Calcul du rayon r = dist(C, u_n)
    r = np.sqrt((ux_n - cx)**2 + (uy_n - cy)**2)
```

```
    # 4. Nouveau point u_{n+1} (l'abscisse est le rayon r)
    un_plus_1 = (r, target_y)
    u_coords.append(un_plus_1)

    # Ajouter la ligne au tableau
    table_data.append([
        n,
        f"({ux_n:.2f}, {uy_n})",
        f"({cx}, {cy})",
        f"{r:.4f}",
        f"({r:.2f}, {target_y})"
    ])

    # --- Graphique ---
    # Tracer l'arc de cercle
    theta = np.linspace(0, np.pi/2, 50)
    if cy == 0: # Arc de y=1 vers y=0
        arc_x = r * np.cos(theta)
        arc_y = r * np.sin(theta)
    else:       # Arc de y=0 vers y=1
        arc_x = r * np.cos(theta)
        arc_y = 1 - r * np.sin(theta)

    ax.plot(arc_x, arc_y, 'gray', lw=0.8, linestyle='--')
    ax.scatter(r, target_y, color='red', s=30, zorder=5)
    ax.annotate(rf'$\sqrt{{{n+1}}}$', (r, target_y), xytext=(0, 10 if target_y else -15),
                textcoords='offset points', ha='center')


# Affichage du tableau avec tabulate
headers = ["n", "u_n (départ)", "Pivot C", "Rayon r", "u_{n+1} (arrivée)"]
print(tabulate(table_data, headers=headers, tablefmt="grid"))

# Dessin des lignes d'appui et pivots
ax.axhline(0, color='black', lw=1.5, label='y=0')
ax.axhline(1, color='black', lw=1.5, label='y=1')
ax.scatter([0, 0], [0, 1], color='blue', marker='x', label='Pivots $C_0, C_1$')

ax.set_title("Construction géométrique de $u_n = (\sqrt{n}, y)$")
ax.legend(loc='upper left')
plt.grid(alpha=0.3)
plt.show()
```

```
+-----+----------------+-----------+-----------+---------------------+
|   n | u_n (départ)   | Pivot C   |   Rayon r | u_{n+1} (arrivée)   |
+=====+================+===========+===========+=====================+
|   0 | (0.00, 1)      | (0, 0)    |    1      | (1.00, 0)           |
+-----+----------------+-----------+-----------+---------------------+
|   1 | (1.00, 0)      | (0, 1)    |    1.4142 | (1.41, 1)           |
+-----+----------------+-----------+-----------+---------------------+
|   2 | (1.41, 1)      | (0, 0)    |    1.7321 | (1.73, 0)           |
+-----+----------------+-----------+-----------+---------------------+
|   3 | (1.73, 0)      | (0, 1)    |    2      | (2.00, 1)           |
+-----+----------------+-----------+-----------+---------------------+
|   4 | (2.00, 1)      | (0, 0)    |    2.2361 | (2.24, 0)           |
+-----+----------------+-----------+-----------+---------------------+
|   5 | (2.24, 0)      | (0, 1)    |    2.4495 | (2.45, 1)           |
+-----+----------------+-----------+-----------+---------------------+
```

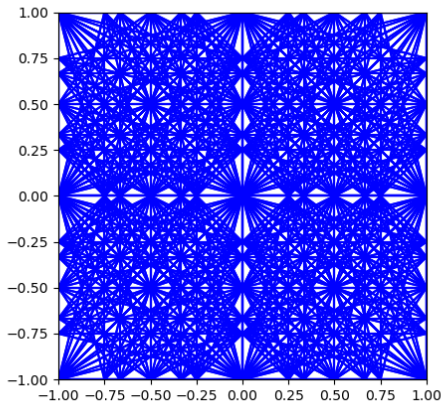


Exercice 8.9 (Diagramme de Farey)

Tracer dans le plan cartésien toutes les droites d'équation $ux + vy = w$ où u, v et w sont des entiers compris entre -4 et 4, et x et y varient dans l'intervalle $[-1; 1]$.

Correction

```
import matplotlib.pyplot as plt
import numpy as np

uu = np.linspace(-4, 4, 9)
vv = np.linspace(-4, 4, 9)
ww = np.linspace(-4, 4, 9)

for w in ww:
    for u in uu:
        for v in vv:
            if v != 0:
                xx = np.array([-1, 1])
                yy = (w - u * xx) / v
                plt.plot(xx, yy, 'b')
            elif u != 0:
                yy = np.array([-1, 1])
                xx = (w - v * yy) / u
                plt.plot(xx, yy, 'b')

plt.xlim(-1, 1)
plt.ylim(-1, 1)
plt.gca().set_aspect('equal', 'box')
plt.savefig('Images/exo-farey.png')
plt.show()
```



Exercice 8.10 (CodEx : Triangle mystérieux)

On souhaite construire une figure géométrique à l'aide d'un processus itératif, en répétant les mêmes étapes de construction plusieurs fois. La démarche est la suivante :

- On considère trois points A, B et C formant un triangle équilatéral.
- On place un point P de façon aléatoire à l'intérieur de ce triangle.
- On répète ensuite les étapes suivantes 100 fois :
  1. Choisir aléatoirement un des sommets A, B ou C.
  2. Calculer le point milieu du segment entre P et le sommet choisi.
  3. Remplacer P par ce nouveau point milieu.

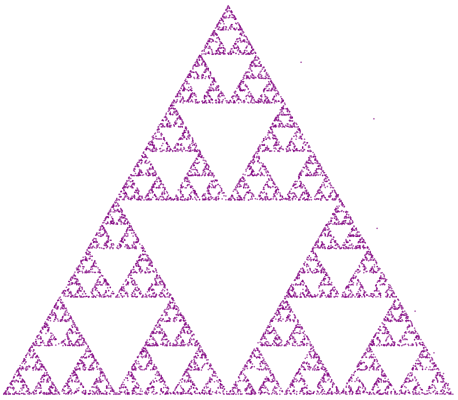
En répétant ces étapes, une jolie figure géométrique se dessine (il s'agit d'une figure fractale). Essayez d'augmenter le nombre de répétitions pour mieux la visualiser.

Source : https://codex.forge.apps.education.fr/exercices/triangle_mystere/#version-%C3%A0-compl%C3%A9ter_1

Correction

La figure obtenue est le *triangle de Sierpiński*.

```python
import matplotlib.pyplot as plt
from random import uniform, choice

# Sommet du triangle équilatéral
A = (0, 0)
B = (1, 0)
C = (0.5, (3**0.5) / 2)

# Point de départ aléatoire
P = (uniform(0, 1), uniform(0, (3**0.5) / 2))

# Nombre d'itérations
iterations = 10000

# Listes pour stocker les coordonnées x et y des points
x_coords = [A[0], B[0], C[0], P[0]]
y_coords = [A[1], B[1], C[1], P[1]]

# Processus itératif
for _ in range(iterations):
    sommet = choice([A, B, C])  # Choisir aléatoirement un sommet
    # Calcul du milieu
    P = ((P[0] + sommet[0]) / 2, (P[1] + sommet[1]) / 2)
    # Ajout des coordonnées du point
    x_coords.append(P[0])
    y_coords.append(P[1])

# Tracé de la figure
plt.figure(figsize=(8, 8))
plt.scatter(x_coords, y_coords, color="purple", s=0.1)
plt.axis("equal")
plt.axis("off")
plt.savefig('Images/Sierpinski.png', dpi=300, bbox_inches='tight')
plt.show()
```



Exercice 8.11 (Module `scipy` - Calcul approché d'une intégrale)
Utiliser `scipy.integrate.quad` pour évaluer l'intégrale

$$\int_0^6 f(x)\,\mathrm{d}x \quad \text{avec} \quad f(x) = \lfloor x \rfloor - 2\left\lfloor \frac{x}{2} \right\rfloor.$$

Après avoir tracé le graphe de la fonction f sur l'intervalle $[0,6]$, calculer la valeur exacte de l'intégrale et la comparer à la valeur approchée obtenue.

Source : https://scipython.com/book2/chapter-8-scipy/questions/numerical-integration-of-a-simple-function/

Correction
La fonction f se réécrit comme

$$f(x) = \begin{cases} 0 & \text{si } 2k < x < 2k+1 \\ 1 & \text{sinon} \end{cases}$$

pour $k \in \mathbb{Z}$. L'intégrale cherchée vaut donc 3.

Pour aider l'intégration numérique, il pourrait être utile de diviser l'intervalle d'intégration en plusieurs sous-intervalles là où la fonction présente des changements brusques ou des points problématiques, puis d'intégrer séparément sur ces sous-intervalles.

```
import numpy as np
func = lambda x: np.floor(x) - 2*np.floor(x/2)

# import matplotlib.pyplot as plt
# xx = np.linspace(0,6,100)
# plt.plot(xx,func(xx),".")
# plt.show()

from scipy.integrate import quad
intervals = [(i, i + 1) for i in range(6)]
val = sum(quad(func, a, b)[0] for a, b in intervals)
print(f"L'integrale vaut approximativement {val:.2f}.")
```

```
L'integrale vaut approximativement 3.00.
```

Exercice 8.12 (Dépréciation ordinateur)

On achète un ordinateur portable à 430 €. On estime qu'une fois sorti du magasin sa valeur u_n en euro après n mois est donnée par la formule

$$u_n = 40 + 300 \times (0.95)^n.$$

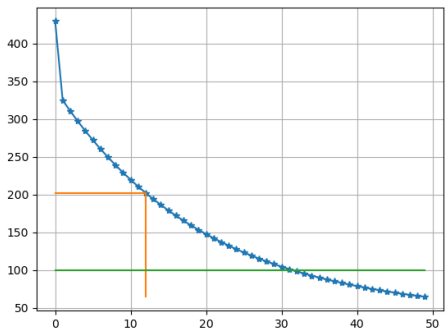
- Que vaut l'ordinateur à la sortie du magasin ? (afficher u_0)
- Que vaut après un an de l'achat ? (afficher u_{12})
- À long terme, à quel prix peut-on espérer revendre cet ordinateur ?
- Déterminer le mois à partir duquel l'ordinateur aura une valeur inférieure à 100 €.

Correction

- À la sortie du magasin $u_0 = 340$
- Après un an de l'achat on a $u_{12} = 40 + 300 \times (0.95)^{12} = 202.11$
- À long terme, on peut espérer revendre cet ordinateur à $\lim_{n\to+\infty} u_n = 40$.
- À partir du 32-ème mois l'ordinateur aura une valeur inférieure à 100 € car :

$$\begin{aligned} 40 + 300 \times (0.95)^n < 100 &\iff (0.95)^n < \frac{100-40}{300} = \frac{1}{5} = 5^{-1} \\ &\iff n\ln(0.95) < -\ln(5) \iff n > -\frac{\ln(5)}{\ln(0.95)} \simeq 31.377 \end{aligned}$$

Vérifions nos calculs :




```
import matplotlib.pyplot as plt
import numpy as np
nn = range(50)
uu = [430]+[40+300*0.95**n for n in nn[1:]]
i = ([u>100 for u in uu]).count(True)
print(f"u[{i-1}]={uu[i-1]} et u[{i}]={uu[i]}")
plt.plot(nn,uu,'*-')
plt.plot([nn[0],nn[12],nn[12]],
    [uu[12],uu[12],uu[-1]])
plt.plot([nn[0],nn[-1]],[100,100])
plt.grid()
plt.show()
```

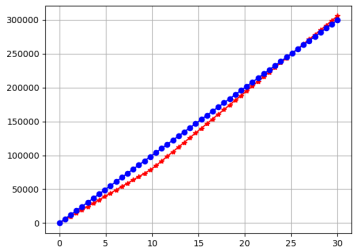
```
u[31]=101.17204772373711 et u[32]=98.11344533755026
```

Exercice 8.13 (Mercato)

Pendant le mercato, un attaquant est courtisé par deux clubs, F et G, qui lui proposent le même salaire mensuel. Cependant, leurs systèmes de primes par but marqué diffèrent : le club F propose une prime de 8 000 euros par but pour les dix premiers buts, puis de 11 300 euros par but à partir du onzième ; le club G offre une prime fixe de 10 000 euros par but, quel que soit le nombre de buts marqués.

1. Écrire les fonctions `F(n)` et `G(n)` qui renvoient respectivement les montants des primes offertes par les clubs F et G en fonction du nombre n de buts marqués.
2. Avec une boucle `while` déterminer le nombre de buts que doit marquer l'attaquant pour que le montant de la prime offerte par le club F soit la plus intéressante.
3. Tracer le graphe des deux fonctions pour vérifier le résultat.
4. Demander au module `scipy` de calculer la solution de $F(n) = G(n)$.

Correction



On a les deux fonctions

$$F(b) = \begin{cases} 8000b & \text{si } b \leq 10, \\ 11300(b-10) + 8000 \times 10 & \text{sinon ;} \end{cases}$$

$$G(b) = 10000b.$$

```
F = lambda b : (b*8000) if (b<=10) else (8000*10+11300*(b-10))
G = lambda b : 10000*b

n=1
while F(n)<G(n):
    n+=1
print(n)

import matplotlib.pyplot as plt
import numpy as np
nn = np.linspace(0,30)
ff = [F(b) for b in nn]
gg = [G(b) for b in nn]
plt.figure(num=None, figsize=(20, 10))
plt.plot(nn,ff,'r*-',nn,gg,'bo-')
plt.grid()
plt.show()

from scipy.optimize import fsolve
print(fsolve(lambda n:F(n)-G(n),20))
```

26 [25.38461538]

Exercice 8.14 (Résolution graphique d'une équation)

Soit la fonction

$$f : [-10, 10] \to \mathbb{R}$$
$$x \mapsto \frac{x^3 \cos(x) + x^2 - x + 1}{x^4 - \sqrt{3}x^2 + 127}.$$

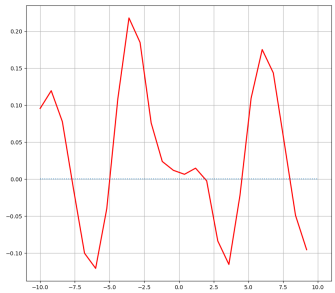
1. Tracer le graphe de la fonction f en utilisant uniquement les valeurs de $f(x)$ lorsque x prend successivement les valeurs $-10, -9.2, -8.4, \ldots, 8.4, 9.2, 10$ (*i.e.* avec un pas de 0.8).
2. Le graphe suggère que l'équation $f(x) = 0$ possède une solution α voisine de 2. En effectuant un zoom autour de cette zone, proposer une valeur approchée de α.

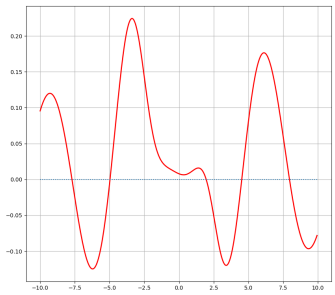
3. Tracer de nouveau le graphe de f en prenant cette fois un pas de 0.05. Cette nouvelle représentation conduit-elle à corriger la valeur approchée de α ?
4. Utiliser le module `scipy` pour obtenir une valeur numérique précise de α.

Correction

On trace d'abord le graphe de $y = f(x)$ ainsi que la droite $y = 0$. Avec un pas de 0.8, la courbe suggère que $\alpha = 1.89$. Avec un pas de 0.05, le tracé est plus fin et on obtient $\alpha = 1.965$. Enfin, en utilisant la fonction `fsolve` de `scipy.optimize`, on trouve $\alpha = 1.96289995$.

```
import matplotlib.pyplot as plt
from numpy import cos, sqrt,arange

f = lambda x: (x**3*cos(x)+x**2-x+1)/(x**4-sqrt(3)*x**2+127)

fig,ax = plt.subplots(2,1,figsize=(10, 20))

xx = arange(-10,10,0.8)
yy = f(xx)
ax[0].plot(xx,yy,'r-',lw=2)
ax[0].plot([-10,10],[0,0],':')
ax[0].grid()

xx = arange(-10,10,0.05)
yy = f(xx)
ax[1].plot(xx,yy,'r-',lw=2)
ax[1].plot([-10,10],[0,0],':')
ax[1].grid()

plt.show()

from scipy.optimize import fsolve
print(fsolve(f,1.9))
```



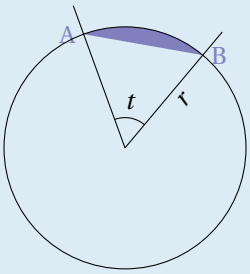


[1.96289995]

Exercice 8.15 (Résolution graphique)

Considérons un cercle ayant un rayon r. Lorsque nous traçons un angle t (mesuré en radians) à partir du centre du cercle, les deux rayons formant cet angle intersectent le cercle en deux points, A et B. Nous définissons a comme l'aire délimitée par la corde et l'arc AB (représentée en bleu sur le schéma). Cette aire est donnée par

$$a = \frac{r^2}{2}(t - \sin(t)).$$



Maintenant, pour un cercle donné (c'est-à-dire un rayon donné), supposons que nous sélectionnions une certaine aire a (représentant la partie en bleu). Nous cherchons à déterminer quelle valeur de l'angle t permet d'obtenir cette aire spécifique. En d'autres termes, connaissant les valeurs de a et r, notre objectif est de trouver l'angle t qui résout l'équation suivante :

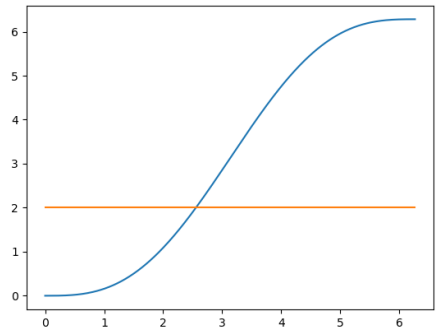
$$\frac{2a}{r^2} = t - \sin(t).$$

1. Résoudre graphiquement l'équation en traçant les courbes correspondant aux membres gauche et droit de l'équation (pour $a = 4$ et $r = 2$). Quelle valeur de t est solution de l'équation ?
2. Comment faire pour obtenir une valeur plus précise du résultat ?

Correction

```
import matplotlib.pyplot as plt
import numpy as np
a, r = 4, 2
tt = np.arange(0,2*np.pi,np.pi/180)
rhs = tt-np.sin(tt)
# rhs = [t-np.sin(t) for t in tt]
lhs = [2*a/r**2 for t in tt]
plt.plot(tt,rhs,tt,lhs)
plt.show()

from scipy.optimize import fsolve
f = lambda t: t-np.sin(t)-2*a/r**2
print(fsolve(f,2.5))
```

[2.55419595]



Le graphe montre que la solution est entre 2 et 3. On peut alors calculer une solution approchée avec `fsolve`.

Attention

Jusqu'ici nous avons représenté des courbes engendrées par des équations cartésiennes, c'est-à-dire des fonctions de la forme $y = f(x)$. Pour cela, nous générons d'abord un ensemble de valeurs $\{x_i\}_{i=0\ldots N}$ puis l'ensemble de valeurs $\{y_i\}_{i=0\ldots N}$ avec $y_i = f(x_i)$. Il existe d'autres types de courbes comme par exemple les courbes paramétrées (engendrées par des équations paramétriques). Les équations paramétriques de courbes planes sont de la forme

$$\begin{cases} x = u(t), \\ y = v(t), \end{cases}$$

où u et v sont deux fonctions cartésiennes et le couple $(x; y)$ représente les coordonnées d'un point de la courbe paramétrée. La courbe engendrée par l'équation cartésienne $y = f(x)$ est une courbe paramétrée car il suffit de poser $u(t) = t$ et $v(t) = f(t)$. Un type particulier de courbe paramétrique est constitué par les équations polaires de courbes planes qui sont de la forme [a]

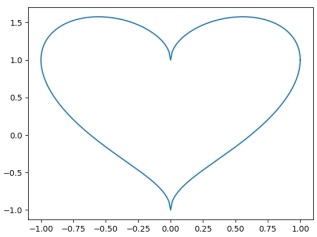
$$\begin{cases} x = r(t)\cos(t), \\ y = r(t)\sin(t). \end{cases}$$

a. r représente la distance de l'origine O du repère $(O; \mathbf{i}, \mathbf{j})$ au point $(x; y)$ de la courbe, et t l'angle avec l'axe des abscisses.

Exercice 8.16 (Courbe paramétrée)

Tracer la courbe mystère suivante pour $t \in [0; 2\pi]$:

$$\begin{cases} x(t) = \cos(t) \\ y(t) = \sin(t) + \sqrt{|\cos(t)|} \end{cases}$$



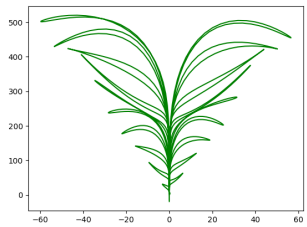

```
import matplotlib.pyplot as plt
import numpy as np
x = lambda t : np.cos(t)
y = lambda t : np.sin(t)+np.sqrt(abs(np.cos(t)))
tt = np.linspace(0,2*np.pi,501)
xx = [x(t) for t in tt]
yy = [y(t) for t in tt]
plt.plot(xx,yy)
# plt.fill(xx,yy,color='pink')
plt.show()
```

Notons qu'on peut écrire directement `xx = x(tt)` et `yy = y(tt)` car les fonctions $t \mapsto x(t)$ et $t \mapsto y(t)$ sont des fonctions définies par composition de fonctions `numpy` qui sont donc vectorisées.

Exercice 8.17 (Courbe paramétrée)

Tracer la courbe mystère suivante pour $t \in [0;39\pi/2]$:

$$\begin{cases} x(t) = t\cos^3(t) \\ y(t) = 9t\sqrt{|\cos(t)|} + t\sin(t/5)\cos(4t) \end{cases}$$

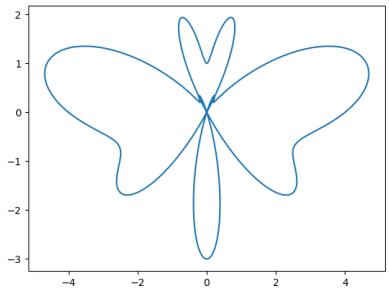



```
from numpy import linspace, cos, sin, sqrt, pi
import matplotlib.pyplot as plt
tt = linspace(0,39*pi/2,1000)
xx = tt*cos(tt)**3
yy = 9*tt*sqrt(abs(cos(tt)))+tt*sin(0.2*tt)*cos(4*tt)
plt.plot(xx,yy,c="green")
plt.show();
```

Exercice 8.18 (Courbe polaire)

Tracer la courbe papillon ($t \in [0;2\pi]$) :

$$\begin{cases} x(t) = r(t)\cos(t) \\ y(t) = r(t)\sin(t) \end{cases} \quad \text{avec } r(t) = \sin(7t) - 1 - 3\cos(2t).$$




```
import matplotlib.pyplot as plt
import numpy as np
r = lambda t : np.sin(7*t)-1-3*np.cos(2*t)
x = lambda t : r(t)*np.cos(t)
y = lambda t : r(t)*np.sin(t)
tt = np.linspace(0,2*np.pi,1001)
xx = [x(t) for t in tt]
yy = [y(t) for t in tt]
plt.plot(xx,yy)
plt.show()
```

La fonction $t \mapsto r(t)$ est une fonction définie par composition de fonctions `numpy` qui sont vectorisées, on peut alors écrire directement

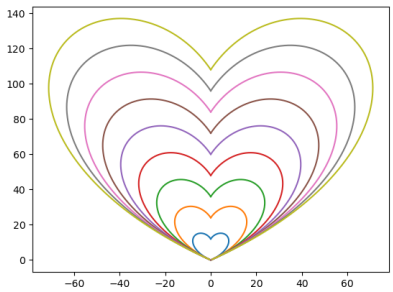
```
import matplotlib.pyplot as plt
import numpy as np
r = lambda t : np.sin(7*t)-1-3*np.cos(2*t)
tt = np.linspace(0,2*np.pi,1001)
xx = r(tt)*np.cos(tt)
yy = r(tt)*np.sin(tt)
plt.plot(xx,yy)
plt.show()
```

Exercice 8.19 (Courbe polaire)

Pour $h = \frac{1}{10}$ et $k = h, 2h, \ldots, 1$, tracer la courbe suivante ($t \in [0;60]$) :

$$\begin{cases} x(t) = \pm r(t)\sin(\pi t/180) \\ y(t) = r(t)\cos(\pi t/180) \end{cases} \quad \text{avec } r(t) = kh(-t^2 + 40t + 1200).$$

Correction




```
from numpy import cos, sin, sqrt, pi, arange
import matplotlib.pyplot as plt
h = 0.1
tt = arange(0,60,h)
idx = 0
for k in arange(h,1,h):
    rr = k*h*(-tt**2+40*tt+1200)
    xx_R =  rr*sin(pi*tt/180)
    yy_R =  rr*cos(pi*tt/180)
    xx_L = -rr*sin(pi*tt/180)
    yy_L = yy_R
    plt.plot(xx_L,yy_L,color='C'+str(idx))
    plt.plot(xx_R,yy_R,color='C'+str(idx))
    idx += 1
plt.show()
```

Exercice 8.20 (Proies et prédateurs)

Le modèle de Volterra, développé par le mathématicien Vito Volterra en 1926, décrit les fluctuations des populations de deux espèces interagissant, comme les proies et leurs prédateurs, telles que les sardines et les requins dans l'Adriatique. Dans ce modèle, les populations de proies et de prédateurs varient dans le temps en fonction de plusieurs facteurs, tels que la reproduction des proies, la prédation par les prédateurs et la reproduction des prédateurs en fonction des proies capturées. Les équations différentielles qui modélisent ce système sont

$$\begin{cases} \frac{d}{dt}e(t) = \big(\mathrm{A} - \mathrm{B}c(t)\big)e(t), & \text{évolution des proies,} \\ \frac{d}{dt}c(t) = \big(-\mathrm{C} + \mathrm{D}e(t)\big)c(t), & \text{évolution des prédateurs,} \end{cases}$$

où $e(t)$ représente la population des proies à l'instant t, et $c(t)$ celle des prédateurs. Les paramètres A, B, C et D sont des constantes caractéristiques des espèces étudiées : A est le taux de reproduction des proies, B le taux de mortalité des proies par prédation, C le taux de mortalité des prédateurs, et D le taux de reproduction des prédateurs en fonction des proies capturées.

Pour effectuer des simulations numériques sur ce modèle, on de le discrétiser à l'aide d'un schéma numérique, comme le schéma d'Euler explicite. Ce schéma permet de calculer les valeurs successives de e_{n+1} et c_{n+1} à partir des valeurs précédentes e_n et c_n en utilisant une approche itérative. Le schéma d'Euler appliqué à ce modèle donne les équations discrètes suivantes

$$\begin{cases} e_{n+1} = e_n\,(1 + \mathrm{A} - \mathrm{B}c_n)\,, & \text{évolution des proies,} \\ c_{n+1} = c_n\,(1 - \mathrm{C} + \mathrm{D}e_n)\,, & \text{évolution des prédateurs.} \end{cases}$$

Ces équations discrétisées nous permettent de simuler l'évolution des populations de proies et de prédateurs à chaque étape n, où e_n et c_n sont respectivement les populations des proies et des prédateurs à l'instant discret n.

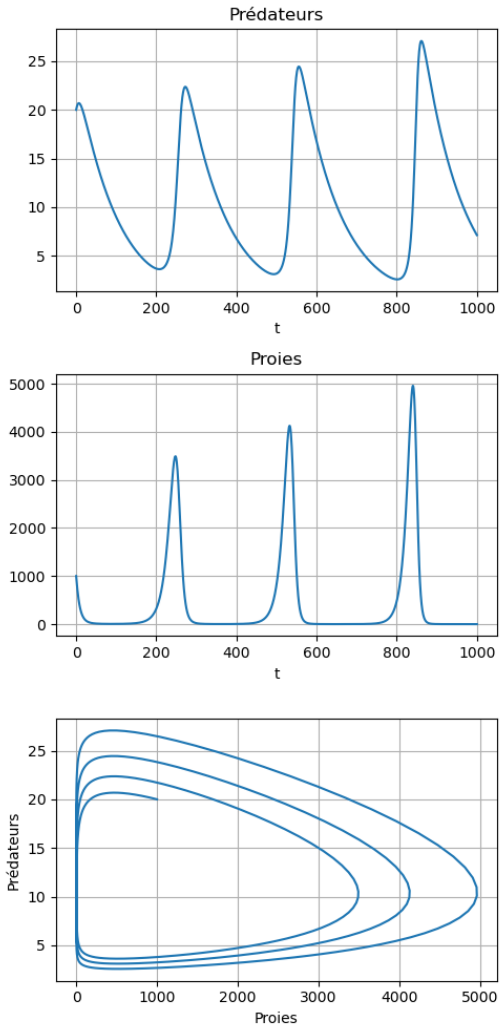
Dans cet exercice, nous allons appliquer ce modèle pour afficher l'évolution des populations de proies et de prédateurs entre $n = 0$ et $n = 1000$, en partant des conditions initiales suivantes :

$$e_0 = 1000, \quad c_0 = 20, \quad \mathrm{A} = 0.1, \quad \mathrm{B} = \mathrm{C} = 0.01, \quad \mathrm{D} = 0.00002.$$

Calculer et afficher l'évolution de ces populations à chaque étape.

Correction

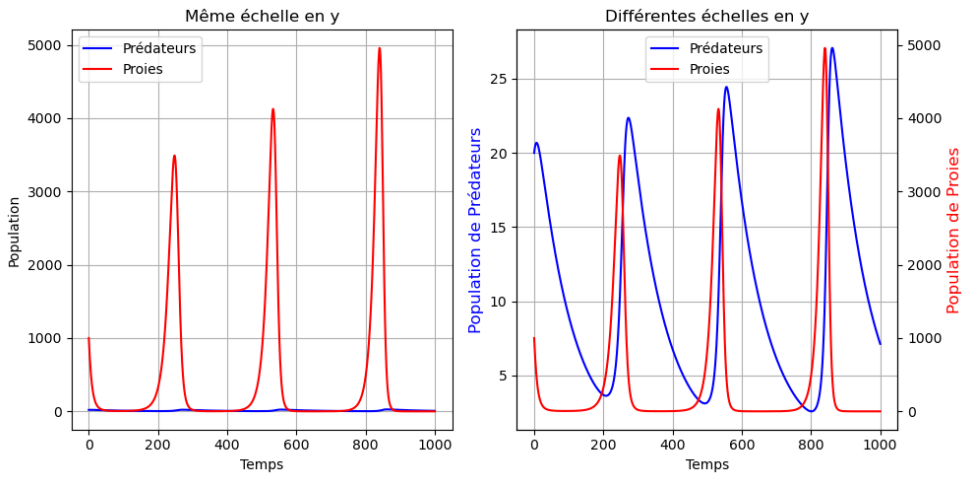
```python
import matplotlib.pyplot as plt
import numpy as np

e,c = 1000,20
Er, Ca = [e], [c]
A,B,C,D = 0.1,0.01,0.01,0.00002
for n in range(1000):
    e, c =  e*(1+A-B*c), c*(1-C+D*e)
    Ca.append(c)
    Er.append(e)

fig,ax = plt.subplots(3,1, figsize=(5,10))

ax[0].plot(range(len(Ca)),Ca)
ax[0].set_title('Prédateurs')
ax[0].set_xlabel('t')
ax[0].grid()

ax[1].plot(range(len(Er)),Er)
ax[1].set_title('Proies')
ax[1].set_xlabel('t')
ax[1].grid()

ax[2].plot(Er,Ca)
ax[2].set_xlabel("Proies")
ax[2].set_ylabel("Prédateurs")
ax[2].grid()

# Ajuste les espacements entre les sous-graphes
plt.tight_layout()
plt.show()
```



Bonus : il serait intéressant d'afficher les deux premières courbes dans un même repère. Cependant, elles ont des échelles très différentes en ordonnées. Nous allons donc utiliser deux systèmes d'axes dans un même graphique. Cela se fait en invoquant la méthode `twinx`. Notez la possibilité d'afficher les labels pour nos ordonnées en couleur : cela permet de mettre en relation l'échelle proposée avec la bonne courbe associée.

```python
import matplotlib.pyplot as plt
import numpy as np

e, c = 1000, 20
Er, Ca = [e], [c]
A, B, C, D = 0.1, 0.01, 0.01, 0.00002
for n in range(1000):
    e, c =  e * (1 + A - B * c), c * (1 - C + D * e)
    Ca.append(c)
    Er.append(e)

fig, ax = plt.subplots(1, 2, figsize=(10, 5))

# Premier graphique avec une seule échelle en y
ax[0].plot(range(len(Ca)), Ca, label='Prédateurs', color='blue')
ax[0].plot(range(len(Er)), Er, label='Proies', color='red')
ax[0].set_xlabel('Temps')
ax[0].set_ylabel('Population')
ax[0].grid()
ax[0].set_title('Même échelle en y')
ax[0].legend()

# Deuxième graphique avec deux échelles en y
```

```
ax[1].plot(range(len(Ca)), Ca, color='blue')
ax[1].set_ylabel("Population de Prédateurs", color="blue", fontsize=12)
ax2 = ax[1].twinx()
ax2.plot(range(len(Er)), Er, color='red')
ax2.set_ylabel("Population de Proies", color="red", fontsize=12)
ax[1].set_xlabel('Temps')
ax[1].grid()
ax[1].set_title('Différentes échelles en y')

# Préparation et affichage de la légende pour les deux courbes
lines = [ax[1].get_lines()[0], ax2.get_lines()[0]]
plt.legend(lines, ["Prédateurs", "Proies"], loc="upper center")

# Ajuste les espacements entre les sous-graphes
plt.tight_layout()
plt.show()
```



Exercice 8.21 (Coïncidences et anniversaires)

Combien faut-il réunir de personne pour avoir une chance sur deux que deux d'entre elles aient le même anniversaire ?

Au lieu de nous intéresser à la probabilité que cet événement se produise, on va plutôt s'intéresser à l'événement inverse : quelle est la probabilité pour que n personnes n'aient pas d'anniversaire en commun (on va oublier les années bissextiles et le fait que plus d'enfants naissent neuf mois après le premier de l'an que neuf mois après la Toussaint.)

- si $n = 1$ la probabilité est 1 (100%) : puisqu'il n'y a qu'une personne dans la salle, il y a 1 chance sur 1 pour qu'elle n'ait pas son anniversaire en commun avec quelqu'un d'autre dans la salle (puisque, fatalement, elle est toute seule dans la salle) ;
- si $n = 2$ la probabilité est $\frac{364}{365}$ ($= 99{,}73\%$) : la deuxième personne qui entre dans la salle a 364 chances sur 365 pour qu'elle n'ait pas son anniversaire en commun avec la seule autre personne dans la salle ;
- si $n = 3$ la probabilité est $\frac{364}{365} \times \frac{363}{365}$ ($= 99{,}18\%$) : la troisième personne qui entre dans la salle a 363 chances sur 365 pour qu'elle n'ait pas son anniversaire en commun avec les deux autres personnes dans la salle mais cela sachant que les deux premiers n'ont pas le même anniversaire non plus, puisque la probabilité pour que les deux premiers n'aient pas d'anniversaire en commun est de 364/365, celle pour que les 3 n'aient pas d'anniversaire commun est donc $364/365 \times 363/365$ et ainsi de suite ;
- si $n = k$ la probabilité est $\dfrac{364}{365} \times \dfrac{363}{365} \times \dfrac{362}{365} \times \cdots \times \dfrac{365-k+1}{365}$.

On obtient la formule de récurrence

$$\begin{cases} \mathrm{P}_1 = 1, \\ \mathrm{P}_{k+1} = \dfrac{365-k+1}{365}\mathrm{P}_k. \end{cases}$$

1. Tracer un graphe qui affiche la probabilité que deux personnes ont la même date de naissance en fonction du nombre de personnes.
2. Calculer pour quel k on passe sous la barre des 50%.

Source : blog http://eljjdx.canalblog.com/archives/2007/01/14/3691670.html et https://calmcode.io/birthday-problem/birthdays.html

Correction
La probabilité que deux personnes dans un groupe de k personne n'ont pas la même date de naissance est

$$\mathrm{P}_{\text{same}}(k) = 1 - \mathrm{P}_{\text{no overlap}}(k) \qquad \text{et} \qquad \mathrm{P}_{\text{no overlap}}(k) = \frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \cdots \times \frac{365-k+1}{365}.$$

```
from math import prod
k = 3
nP = prod([(365-i+1)/365 for i in range(2,k+1)])
P = 1-nP
print(f"{k = }, {nP = }, {P = }")
```

```
k = 3, nP = 0.9917958341152187, P = 0.008204165884781345
```

Dans un groupe de 23 personnes (=`indice`), il y a plus d'une chance sur deux (`seuil=0.5`) pour que deux personnes de ce groupe aient leur anniversaire le même jour (`tot=365`). Ou, dit autrement, il est plus surprenant de ne pas avoir deux personnes qui ont leur anniversaire le même jour que d'avoir deux personnes qui ont leur anniversaire le même jour (et avec 57, on dépasse les 99% de chances !)

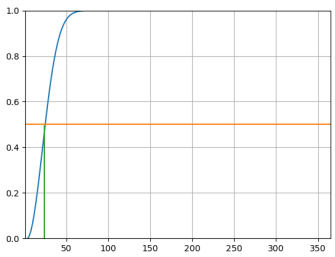
```
from math import prod

def calculate(k):
    nP = prod([(365-i+1)/365 for i in range(2,k+1)])
    P = 1-nP
    return nP, P

k = 23
nP, P = calculate(k)
print(f"{k = }, {nP = }, {P = }")
```

```
k = 23, nP = 0.4927027656760144, P = 0.5072972343239857
```

Cas générale (pour un groupe de $k \in [1;365]$ personne) avec détection du nombre de personnes pour dépasser un seuil imposé :

```
tot, seuil = 365, 0.5
nn = range(1,tot)
P = [1]
for k in range(tot-2):
    P.append( (tot-k+1)*P[k]/tot )
nP = [1-p for p in P]
indice = ([np<seuil for np in nP]).count(True)
print(f"Dans un groupe de {indice-1} personnes on a {seuil*100}\% pour que deux personnes")
print("de ce groupe aient leur anniversaire le même jour")
plt.plot(nn,nP, [min(nn),max(nn)],[seuil,seuil],[indice,indice],[0,P[indice]])
plt.axis([1,tot,0,1])
plt.grid()
```

Dans un groupe de 23 personnes on a 50.0% pour que deux personnes de ce groupe aient leur anniversaire le même jour



On peut s'amuser à adapter les calculs à d'autres problèmes, par exemple on a `seuil=61%` de chances que parmi `indice=5` personnes prises au hasard, deux ont le même signe astrologique (`tot=12`).

Exercice 8.22 (Conjecture de Syracuse)

Considérons la suite récurrente

$$\begin{cases} u_1 \in \mathbb{N}^* \text{ donné,} \\ u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } n \text{ est pair,} \\ 3u_n + 1 & \text{sinon.} \end{cases} \end{cases}$$

En faisant des tests numérique on remarque que la suite obtenue tombe toujours sur 1 peut importe l'entier choisit [a] au départ. La conjecture de Syracuse affirme que, peu importe le nombre de départ choisi, la suite ainsi construite atteint le chiffre 1 (et donc boucle sur le cycle 4, 2, 1). Cet énoncé porte le nom de «Conjecture» et non de théorème, car ce résultat n'a pas (encore) été démontré pour tous les nombres entiers. En 2004, la conjecture a été "juste" vérifiée pour tous les nombres inférieurs à 2^{64}.

1. Écrire un script qui, pour une valeur de $u_1 \in]1; 10^6]$ donnée, calcule les valeurs de la suite jusqu'à l'apparition du premier 1.
2. Tracer les valeurs de la suite en fonction de leur position (on appelle cela la trajectoire ou le vol), *i.e.* les points $\{(n, u_n)\}_{n=1}^{n=\mathrm{N}}$
3. Calculer ensuite le *durée de vol*, *i.e.* le nombre de terme avant l'apparition du premier 1 ; l'*altitude maximale*, *i.e.* le plus grand terme de la suite et le *facteur d'expansion*, c'est-à-dire l'altitude maximale divisée par le premier terme.

On peut s'amuser à chercher les valeurs de u_1 donnant la plus grande durée de vol ou la plus grandes altitude maximale. On notera que, même en partant de nombre peu élevés, il est possible d'obtenir des altitudes très hautes. Vérifiez que, en partant de 27, elle atteint une altitude maximale de 9232 et une durée de vol de 111. Au contraire, on peut prendre des nombres très grands et voir leur altitude chuter de manière vertigineuse sans jamais voler plus haut que le point de départ. Faire le calcul en partant de 10^6.

Ce problème est couramment appelé Conjecture de Syracuse, mais aussi problème de Syracuse, algorithme de Hasse, problème de Ulam, problème de Kakutani, conjecture de Collatz, conjecture du $3n+1$. Vous pouvez lire l'article de vulgarisation https://automaths.blog/2017/06/20/la-conjecture-de-syracuse/amp/

a. Dès que $u_i = 1$ pour un certain i, la suite devient périodique de valeurs 4, 2, 1

Correction

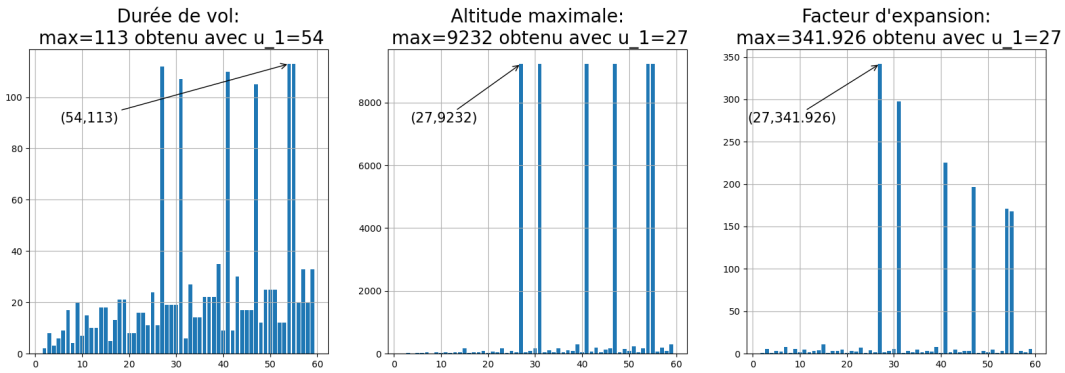
On écrit d'abord une fonction qui prend en entrée la valeur de u_1 et renvoie la suite de Syracuse. Vue que la division par 2 se fait uniquement sur des nombres pairs, la suite reste une suite d'entiers. Pour garder cette propriété, il faut utiliser la division entière :

```
def suite(n):
    u = [n]
    while u[-1] != 1 :
        u.append( u[-1]//2 if u[-1]%2==0 else 3*u[-1]+1 )
    return u
```

On teste cette fonction :

```
import matplotlib.pyplot as plt

N = 60
Uinit = list(range(2, N))

L, M, F = [], [], []
for n in Uinit:
    U = suite(n)
    L.append(len(U))
    M.append(max(U))
    F.append(M[-1] / n)

# Plotting
plt.figure(figsize=(20, 6))
```

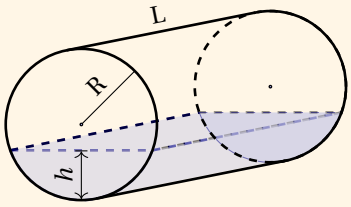
```
# Duration of flight
ax1 = plt.subplot(1, 3, 1)
ax1.bar(Uinit, L)
maxL = max(L)
indicemaxL = L.index(maxL) + 2
# ax1.plot([Uinit[0], indicemaxL, indicemaxL], [maxL, maxL, 0], 'r:')
ax1.set_title(f"Durée de vol:\n max={maxL} obtenu avec u_1={indicemaxL}", size=20)
ax1.annotate(f'({indicemaxL},{maxL})',
             xy=(indicemaxL, maxL), xycoords='data',
             xytext=(0.3, 0.75), textcoords='axes fraction', size=15,
             arrowprops=dict(arrowstyle="->"),
             horizontalalignment='right',
             verticalalignment='bottom')
ax1.grid()

# Maximum altitude
ax2 = plt.subplot(1, 3, 2)
ax2.bar(Uinit, M)
maxM = max(M)
indicemaxM = M.index(maxM) + 2
# ax2.plot([Uinit[0], indicemaxM, indicemaxM], [maxM, maxM, 0], 'r:')
ax2.set_title(f"Altitude maximale:\n max={maxM} obtenu avec u_1={indicemaxM}", size=20)
ax2.annotate(f'({indicemaxM},{maxM})',
             xy=(indicemaxM, maxM), xycoords='data',
             xytext=(0.3, 0.75), textcoords='axes fraction', size=15,
             arrowprops=dict(arrowstyle="->"),
             horizontalalignment='right',
             verticalalignment='bottom')
ax2.grid()

# Expansion factor
ax3 = plt.subplot(1, 3, 3)
ax3.bar(Uinit, F)
maxF = max(F)
indicemaxF = F.index(maxF) + 2
# ax3.plot([Uinit[0], indicemaxF, indicemaxF], [maxF, maxF, 0], 'r:')
ax3.set_title(f"Facteur d'expansion:\n max={maxF:g} obtenu avec u_1={indicemaxF}", size=20)
ax3.annotate(f'({indicemaxF},{maxF:g})',
             xy=(indicemaxF, maxF), xycoords='data',
             xytext=(0.3, 0.75), textcoords='axes fraction', size=15,
             arrowprops=dict(arrowstyle="->"),
             horizontalalignment='right',
             verticalalignment='bottom')
ax3.grid()

plt.show()
```



Exercice Bonus 8.23 (Cuve de fioul enterrée)

J'ai acquis une ancienne maison équipée d'une cuve de stockage de mazout enterrée. La capacité de cette cuve est inconnue, mais je peux mesurer la hauteur du mazout à l'intérieur ainsi que le rayon, qui est de 0.80 mètres. Au départ, la hauteur du mazout dans la cuve était de $h_1 = 0.36$ mètres. Après avoir ajouté 3 000 litres, la hauteur a atteint $h_2 = 1.35$ mètres.

1. Calculer le volume total de la cuve.
2. Tracer une fonction qui donne la quantité de litres que je peux ajouter en fonction de la hauteur du mazout dans la cuve.



Correction

1. **Calcul de la capacité de la cuve.**

   Notons L la longueur de la cuve, R son rayon et h la hauteur du mazout. On a $0 \leq h \leq 2\mathrm{R}$. Considérons une coupe verticale de la cuve comme à la Figure 8.1a.

   L'introduction de 3000 L, c'est-à-dire 3 m^3, de mazout fait passer h de h_1 à h_2. Ce volume correspond à $\mathrm{L} \times \mathscr{A}$ où $\mathscr{A}$ est la surface coloriée de la Figure 8.1b.

   Pour calculer l'aire coloriée, on "tourne" le cercle et on le plonge dans un repère orthonormé comme à la Figure 8.1c.

   On peut alors calculer l'aire grâce au calcul d'une intégrale :

$$\mathscr{A} = 2\int_{h_1}^{h_2} \sqrt{x(2\mathrm{R}-x)}\,\mathrm{d}x.$$

   Comme $\mathrm{L} \times \mathscr{A} = 3\ \mathrm{m}^3$, on trouve ensuite la longueur L de la cuve (en mètres) et le volume totale de la cuve est $\pi\mathrm{R}^2\mathrm{L}$ (en mètres cubes), c'est-à-dire $10^3\pi\mathrm{R}^2\mathrm{L}$ (en litres).

   Le calcul analytique est possible (voir à la fin de cette correction) mais nous pouvons nous appuyer sur un calcul approché :

```python
from numpy import sqrt, pi
from scipy import integrate

R  = 0.8  # en mètres
h1 = 0.36 # en mètres
h2 = 1.35 # en mètres

g = lambda x : sqrt(x*(2*R-x))

A_m2 = 2*integrate.quad(g,h1,h2)[0]  # en mètres carrés
L_m  = 3/A_m2                        # en mètres
V_m3 = pi*R**2*L_m                   # en mètres cubes
V_l  = V_m3*1.e3                     # en litres

print(f'A = {A_m2:g} m^2')
print(f'L = {L_m:g} m')
print(f'V = {V_m3:g} m^3 = {V_l:g} litres')
```

```
A = 1.47136 m^2
L = 2.03893 m
V = 4.09952 m^3 = 4099.52 litres
```

2. **Affichage de la fonction : litres à ajouter en fonction de la hauteur de mazout déjà présent dans la cuve.** Soit h la hauteur de mazout dans la cuve (en mètres) et ℓ les litres qu'on peut ajouter pour remplir la cuve, alors :

$$f : [0;2\mathrm{R}] \to [0;4000]$$
$$h \mapsto 10^3 \times \left(\mathrm{V}_{m^3} - 2 \times \mathrm{L}_m \times \int_0^h \sqrt{x(2\mathrm{R}-x)}\,\mathrm{d}x\right)$$

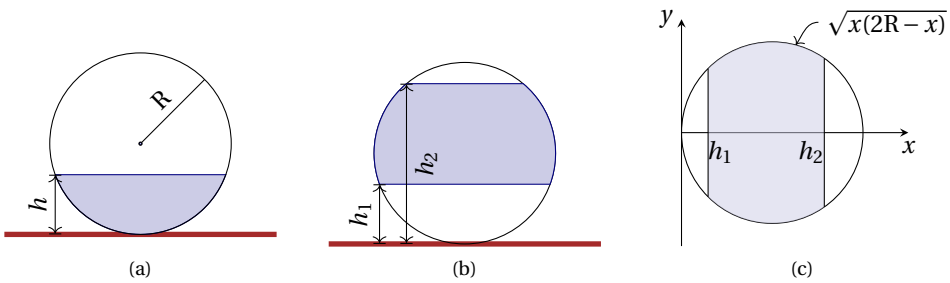
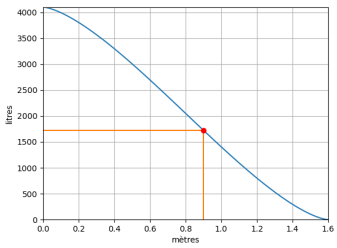


FIGURE 8.1. – Exercice 8.23

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate

f = lambda h : 1.e3*(V_m3-2*L_m*integrate.quad(g,0,h)[0])

hh = np.linspace(0,2*R,101)
yy = [f(h) for h in hh]
plt.plot(hh,yy)

htest = 0.9
ftest = f(htest)
plt.plot([htest,htest,0],[0,ftest,ftest])
plt.plot([htest],[ftest],lw=0.5,marker='o',color='red')
plt.xlabel("mètres")
plt.ylabel("litres")
plt.grid()
plt.axis([0,2*R,0,V_l])
plt.show()
```



$A = 1.47136\ \mathrm{m}^2$ $L = 2.03893\ \mathrm{m}$ $V = 4.09952\ \mathrm{m}^3 = 4099.52$ litres

3. **Calcul analytique** Calculons tous d'abord les primitives de $\sqrt{x(2R-x)}$:

$$\begin{aligned}
\int \sqrt{x(2R-x)}\,\mathrm{d}x &\overset{\substack{x=R-y\\ \mathrm{d}x=-\mathrm{d}y}}{=} -\int \sqrt{R^2-y^2}\,\mathrm{d}y \overset{\substack{y=R\sin(t)\\ \mathrm{d}y=R\cos(t)\,\mathrm{d}t}}{=} -R^2\int \cos^2(t)\,\mathrm{d}t \\
&= -\frac{R^2}{2}(\sin(t)\cos(t)+t)+c = -\frac{R^2}{2}\left(\sin(t)\sqrt{1-\sin^2(t)}+t\right)+c \\
&\overset{t=\arcsin\left(\frac{y}{R}\right)}{=} -\frac{R^2}{2}\left(\frac{y}{R}\sqrt{1-\frac{y^2}{R^2}}+\arcsin\left(\frac{y}{R}\right)\right)+c \\
&\overset{y=R-x}{=} -\frac{R-x}{2}\sqrt{x(2R-x)}-\frac{R^2}{2}\arcsin\left(1-\frac{x}{R}\right)+c.
\end{aligned}$$

Par conséquent

$$\mathscr{A} = 2\int_{h_1}^{h_2} \sqrt{x(2R-x)}\,\mathrm{d}x \approx 1{,}48\,\mathrm{m}^2.$$

```
import sympy as sp
x = sp.Symbol('x', positive=True)
R = sp.Rational(8,10)
Aire = 2*sp.integrate( sp.sqrt(x*(2*R-x)) , (x, sp.Rational(36,100), sp.Rational(135,100)) ) # en
↪  mètres carrés
# L_m  = 3/A              # en mètres
# V_m3 = pi*R**2*L_m      # en mètres cubes
# V_l  = V_m3*10**3       # en litres
print(f"{Aire = } ≈ {Aire.evalf()}")
# print(f'L = {L_m:g} m')
# print(f'V = {V_m3:g} m^3 = {V_l:g} litres')
```

```
Aire = -32*asin(3*sqrt(10)/20)/25 + 33*sqrt(31)/625 + 33*sqrt(15)/400 + 32*asin(3*sqrt(6)/8)/25 ≈
↪ 1.47135828652832
```

Comme $\mathrm{L} \times \mathscr{A} = 3\,\mathrm{m}^3$, on trouve ensuite $\mathrm{L} \approx 2\,\mathrm{m}$ et le volume totale de la cuve est $\pi \mathrm{R}^2 \mathrm{L} \approx 4\,\mathrm{m}^3$, *i.e.* environs 4 000 L. Enfin

$$\begin{aligned} f(h) &= 10^3 \times \left(\mathrm{V}_{m^3} - 2 \times \mathrm{L}_m \times \int_0^h \sqrt{x(2\mathrm{R}-x)}\,\mathrm{d}x\right) \\ &= 10^3 \times \left(\mathrm{V}_{m^3} - 2 \times \mathrm{L}_m \times \left(-\frac{\mathrm{R}-h}{2}\sqrt{h(2\mathrm{R}-h)} - \frac{\mathrm{R}^2}{2}\arcsin\left(1-\frac{h}{\mathrm{R}}\right) + \frac{\mathrm{R}^2}{2}\arcsin(1)\right)\right) \\ &= 10^3 \times \left(\mathrm{V}_{m^3} + \mathrm{L}_m\left((\mathrm{R}-h)\sqrt{h(2\mathrm{R}-h)} + \mathrm{R}^2\arcsin\left(1-\frac{h}{\mathrm{R}}\right) - \frac{\pi\mathrm{R}^2}{2}\right)\right) \end{aligned}$$

- $f(0) = \mathrm{V}_\ell$, $f(2\mathrm{R}) = 0$;
- $f'(h) = -2 \times 10^3 \times \mathrm{L}_m\sqrt{h(2\mathrm{R}-h)}$;
- $f'(h) = 0$ ssi $h = 0$ ou $h = 2\mathrm{R}$;
- f est décroissante pour tout $h \in [0;2\mathrm{R}]$;
- $f''(h) = -2 \times 10^3 \times \mathrm{L}_m \frac{h-\mathrm{R}}{\sqrt{h(2\mathrm{R}-h)}}$;
- $h = \mathrm{R}$ est un point d'inflexion ;
- f convexe pour $h \in [\mathrm{R};2\mathrm{R}]$, concave pour $h \in [0;\mathrm{R}]$.

Exercice Bonus 8.24 (Le crible de MATIYASEVITCH)

Dessinons la parabole d'équation $y = x^2$. Ensuite, on considère deux ensembles de points : les points A_i de coordonnées $(-i, i^2)$ pour tout entier $i \geq 2$, et les points B_j de coordonnées (j, j^2) pour tout entier $j \geq 2$. L'objectif est de relier tous les points A_i aux points B_j et de représenter cette figure à l'aide de `matplotlib`.

En observant cette figure, on constate que tous les segments $[\mathrm{A}_i\mathrm{B}_j]$ croisent l'axe des ordonnées en un point de coordonnées $(0, n)$ où $n \in \mathbb{N}^*$.

Ensuite, on établit une relation entre les nombres premiers et la trajectoire des segments $[\mathrm{A}_i\mathrm{B}_j]$. Plus précisément, on montre qu'un nombre situé sur l'axe des ordonnées n'est pas premier si, et seulement si, l'un des segments $[\mathrm{A}_i\mathrm{B}_j]$ traverse l'axe des ordonnées en ce point.

Finalement, on s'intéresse au nombre de segments qui passent par les points de coordonnées $(0, n)$ avec $n \in \mathbb{N}^*$ et n non premier, et on cherche à comprendre la signification de ce nombre dans le contexte de l'exercice. *Rappel : un nombre $n \in \mathbb{N}^*$ est premier s'il n'est divisible que par 1 et lui même.*

Correction

```python
import matplotlib.pyplot as plt
import numpy as np
# la parabole
xx = np.linspace(-5,5,101)
yy = xx**2
plt.plot(xx,yy)
# les segments
A = [[-i,i**2] for i in range(1,10)]
B = [[ j,j**2] for j in range(1,10)]
for a in A:
    for b in B:
        plt.plot([a[0],b[0]],[a[1],b[1]],'r-')
# fixons les axes pour un meilleur affichage
plt.axis([-5,5,0,19])
plt.xticks(range(-5,5,1), size='small')
plt.yticks(range(0,19,1), size='small')
plt.grid()
plt.show()
```

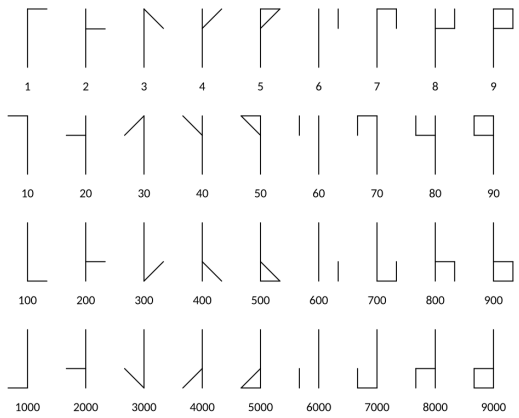
Le segment $[\mathrm{A}_i\mathrm{B}_j]$ appartient à la droite d'équation

$$y = \frac{j^2 - i^2}{j+i}(x-j) + j^2 = (j-i)x + ij$$

et il croise l'axe des ordonnées en le point de coordonnées $(0, ij)$. Comme $i, j \in \mathbb{N}^* \setminus \{1\}$, le produit ij appartient à $\mathbb{N}$.

Un nombre $n \in \mathbb{N}^*$ situé sur l'axe des ordonnées n'est pas premier si, et seulement si, il existe un couple $(i, j) \in \mathbb{N}^* \setminus \{1\}$ tel que $n = ij$ donc si, et seulement si, il existe un couple $(i, j) \in \mathbb{N}^* \setminus \{1\}$ tel que le segment $[\mathrm{A}_i\mathrm{B}_j]$ traverse l'axe des ordonnées en ce point.

Le nombre de segments qui passent par les points de coordonnées $(0, n)$ avec $n \in \mathbb{N}^*$ et n non premier représente le double du nombre de diviseurs propres de n si n n'est pas un carré parfait, sinon c'est le nombre de diviseurs propres de n.



Exercice 8.25 (Notation cistercienne)

Le système de comptage numérique cistercien était utilisé par l'ordre monastique cistercien à la fin de la période médiévale. Les chiffres de 1 à 9 sont représentés par des symboles disposés autour d'une portée verticale (le 0). En plaçant ces symboles réfléchis verticalement et/ou horizontalement à chacun des quatre emplacements, les chiffres décimaux dans les positions unités, dizaines, centaines et milliers pouvaient être représentés, permettant ainsi de définir les nombres de 0 à 9999.

cf. https://en.wikipedia.org/wiki/Cistercian_numerals

Écrire une fonction qui, pour un entier compris entre 0 et 9999, trace sa représentation cistercienne.

Correction

A chaque chiffre on associe un liste de deux tuples : le premier tuple contient les abscisses, le deuxième les ordonnées des points à relier.

```
import matplotlib.pyplot as plt

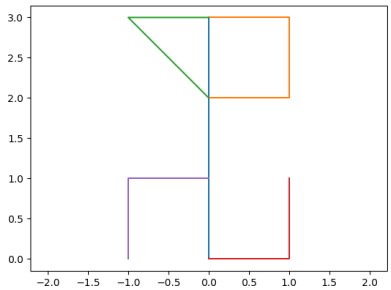
def arabic2cistercian(n):
    base = [(0,0), (0,3)]

    units = { "1": [(0,1),(3,3)] ,      "2": [(0,1),(2,2)] ,
            "3": [(0,1),(3,2)] ,        "4": [(0,1),(2,3)] ,
            "5": [(0,1,0),(3,3,2)] ,    "6": [(1,1),(3,2)] ,
            "7": [(0,1,1),(3,3,2)] ,    "8": [(0,1,1),(2,2,3)] ,
            "9": [(0,1,1,0),(2,2,3,3)]
            }

    tens = { cle: [ tuple( -x for x in val[0]), val[1] ] for cle, val in units.items() }
    hundreds = { cle: [ val[0], tuple( 3-y for y in val[1]) ] for cle, val in units.items() }
    thousands = { cle: [ tuple( -x for x in val[0]), tuple( 3-y for y in val[1]) ] for cle, val in
    ↪  units.items() }

    plt.axis('equal')
    s = str(n)[::-1]
    l = len(s)
    plt.plot( *base )
    if int(s[0])>0: plt.plot( *units[s[0]] )
    if l>1 and int(s[1])>0: plt.plot( *tens[s[1]] )
    if l>2 and int(s[2])>0: plt.plot( *hundreds[s[2]] )
    if l>3 and int(s[3])>0: plt.plot( *thousands[s[3]] )
    plt.show()

arabic2cistercian(8759)
```



Exercice Bonus 8.26 (Taux Marginal, Taux Effectif : comprendre les impôts)

Dans l'imaginaire populaire, passer d'une tranche de revenu à une autre est souvent perçu comme un bouleversement. On entend fréquemment parents et proches s'interroger sur l'effet qu'aurait une augmentation de revenu sur le passage à une tranche supérieure, et donc sur une potentielle augmentation d'impôts.

Pour comprendre ce qui se passe, il est utile de représenter graphiquement l'évolution des impôts en fonction du revenu, ou plutôt du "quotient familial" (QF). En effet, un même revenu n'est pas imposé de la même façon en fonction du nombre de personnes à charge (parts) qu'il est censé soutenir financièrement. Le quotient familial est le résultat de la division du revenu par le nombre de parts, qui est de 1 pour un adulte et de 0.5 pour un enfant, sauf cas particuliers.

Le fonctionnement de l'impôt sur le revenu est le suivant : le revenu imposable pour l'année 2024 est divisé en tranches, et l'impôt à payer en 2025 est calculé en appliquant un pourcentage spécifique à chaque tranche. Voici les tranches de revenus et les taux d'imposition correspondants pour les revenus de 2024 (impôts 2025), d'après les données officielles du ministère des finances : [a]

Tranche de revenu 2024	Jusqu'à 11 497 €	De 11 497 € à 29 315 €	De 29 315 € à 83 823 €	De 83 823 € à 180 294 €	Plus de 180 294 €
Taux d'imposition 2025	0%	11%	30%	41%	45%

Dans ce tableau, le terme "taux d'imposition" désigne en réalité le taux marginal, qui s'applique uniquement à une partie des revenus, à savoir celle de la tranche concernée. On observe que le taux marginal est plus élevé pour les revenus élevés, illustrant ainsi la progressivité de l'impôt. De plus, à l'intérieur de chaque tranche, le montant d'impôt à payer est proportionnel au quotient familial. On dit que la fonction impôts est linéaire par morceaux.

Ces tranches d'imposition se traduisent de la manière suivante :

- Pour un revenu inférieur ou égale à 11 497 €, aucun impôt n'est dû.
- Si le revenu se situe entre 11 498 € et 29 315 €, aucune imposition n'est appliquée sur les premiers 11 497 €, puis un taux de 11% est appliqué sur la partie excédant 11 497 €.
  Par exemple, avec un revenu de 11 498 €, l'impôt sera de 11 centimes car 11% de 1 € c'est 11 centimes.
- Pour un revenu compris entre 29 316 € et 83 823 € :
  - aucun impôt n'est appliqué sur la première tranche de 11 497 € ;
  - puis un taux de 11% est appliqué sur la deuxième tranche, qui va de 11 497 € à 29 315 €, soit 11% de (29 315 − 11 497), c'est-à-dire (29 315 − 11 497) × 0.11 = 1 959,98 € ;
  - enfin, un taux de 30% est appliqué sur la partie du revenu excédant 29 315 €.

  Par exemple, pour un revenu de 72 000 €, l'impôt sera de 0 € (première tranche) + 1 959,98 € (deuxième tranche) + 30% de (72 000 − 29 315) = 12805.5 €, soit un total de 14765.78 €.

Prenons l'exemple d'un célibataire (foyer d'une seule part) qui, en 2024 déclare un revenu net imposable de 100 000 € pour l'année 2022. Calculons explicitement les impôts à payer. Étant donné que ce contribuable se situe dans une tranche d'imposition marginale de 41%, le montant total d'impôts s'élève à :

$$0\% \times (11\,497 - 0) + 11\% \times (29\,315 - 11\,497) + 30\% \times (83\,823 - 29\,315) + 41\% \times (100\,000 - 83\,823) = 24944.95\,€,$$

ce qui représente 24.94495% de ses revenus (et non pas 45% comme cela pourrait être initialement pensé).

Afin de mieux comprendre ces calculs, il est possible de visualiser graphiquement différentes fonctions en fonction du revenu imposable. Cela permettra de vérifier les montants d'impôts obtenus. Écrire et tracer le graphe des fonctions suivantes en fonction du revenu imposable (ou du QF) : 1. Taux d'imposition marginal, 2. Montant de l'impôt, 3. Taux réel d'imposition, 4. Ce qui reste après avoir payé l'impôt.

Vérifier notamment l'exemple donné ici https://www.service-public.fr/particuliers/vosdroits/F1419 : si un célibataire déclare 30 000 €, le montant de l'impôt est €.

a. https://www.service-public.fr/particuliers/vosdroits/F1419

Correction

On commence par comprendre mathématiquement comment les obtenir, puis on écrira le code.

1. Commençons par tracer le graphique responsable de la crainte du saut d'une tranche : en horizontale le revenu imposable, en verticale le taux de la tranche correspondante (appelé *taux marginal)*. Il s'agit d'une fonction constante par morceaux qui présente effectivement des sauts importants. Mais *ces sauts ne concernent pas l'impôt*

mais la dérivée de la fonction Impôt I *par rapport au revenu* R, c'est à dire de la fonction $\mathrm{I}' \colon \mathbb{R}_+ \to \mathbb{R}_+$ définie par

$$\mathrm{I}'(\mathrm{R}) = \begin{cases} 0 & \text{si } \mathrm{R} \in [0, 11497] \\ 0.11 & \text{si } \mathrm{R} \in]11497, 29315] \\ 0.3 & \text{si } \mathrm{R} \in]29315, 83823] \\ 0.41 & \text{si } \mathrm{R} \in]83823, 180294] \\ 0.45 & \text{si } \mathrm{R} \in]180294, +\infty[\end{cases}$$

Ce graphique indique le pourcentage d'imposition qui serait appliqué *sur chaque euro supplémentaire* qui viendrait s'ajouter au revenu.

2. La fonction Impôt $\mathrm{I} \colon \mathbb{R}_+ \to \mathbb{R}_+$ est une fonction continue ! Elle est définie par

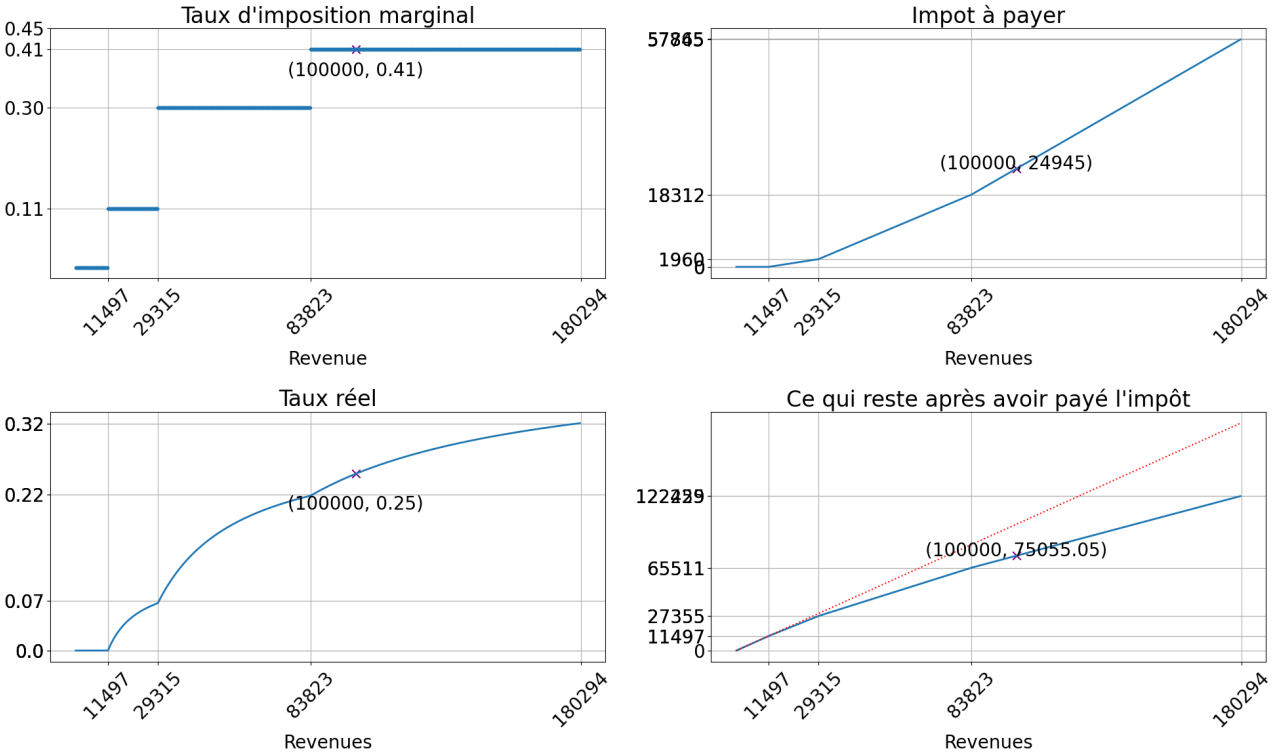
$$\mathrm{I}(\mathrm{R}) = \int_0^{\mathrm{R}} \mathrm{I}'(r)\,\mathrm{d}r = \begin{cases} 0 & \text{si } \mathrm{R} \in [0, 11497] \\ 0 + 0.11(\mathrm{R} - 11497) & \text{si } \mathrm{R} \in]11497, 29315] \\ 0 + 0.11(29315 - 11497) + 0.3(\mathrm{R} - 29315) & \text{si } \mathrm{R} \in]29315, 83823] \\ 0 + 0.11(29315 - 11497) + 0.3(83823 - 29315) + 0.41(\mathrm{R} - 83823) & \text{si } \mathrm{R} \in]83823, 180294] \\ 0 + 0.11(29315 - 11497) + 0.3(83823 - 29315) + 0.41(180294 - 83823) + 0.45(\mathrm{R} - 180294) & \text{si } \mathrm{R} \in]180294, +\infty[\end{cases}$$

Elle est bien une fonction continue du revenu, croissante, affine par morceaux et convexe.

3. Pour calculer le Taux réel d'imposition, c'est-à-dire le pourcentage qu'il faut appliquer au revenu pour avoir l'impôt, il s'agit simplement de diviser I par R. Il s'agit de la fonction $\mathrm{G} \colon \mathbb{R}_+ \to \mathbb{R}_+$ définie par $\mathrm{G}(\mathrm{R}) = \frac{\mathrm{I}(\mathrm{R})}{\mathrm{R}}$. Chaque changement de tranche se repère ici par un point anguleux dans la courbure de la fonction.
4. Ce dernier graphique montre une autre propriété qui mérite d'être signalée car elle n'est pas évidente pour tout le monde. Horizontalement, toujours le revenu. Verticalement, on indique «ce qui reste quand on a payé les impôts», autrement dit la différence entre le revenu et l'impôt. Eh bien, cette fonction est encore croissante. Qu'est-ce que cela signifie ? Tout simplement que plus on gagne et plus on est riche, même après avoir déduit les impôts.

   Il s'agit de la fonction $\mathrm{A} \colon \mathbb{R}_+ \to \mathbb{R}_+$ définie par $\mathrm{A}(\mathrm{R}) = \mathrm{R} - \mathrm{I}(\mathrm{R})$.

Les graphes demandés sont les suivants :



En examinant le graphique des impôts payés en fonction du QF, on constate qu'il n'y a pas de sauts brusques lorsque l'on passe d'une tranche à une autre. Mathématiquement, cela signifie que l'impôt est une *fonction continue* du QF, ce qui garantit une transition en douceur lors des changements de revenu : une petite modification du revenu n'entraîne qu'une petite modification de l'impôt (sauter d'une tranche n'est pas un drame). Avec des taux marginaux par tranches,

on obtient une fonction *croissante* (plus le revenu est élevé et plus l'impôt est élevé) et affine par morceaux; et puisque le taux marginal augmente en fonction du revenu il s'agit d'une fonction *convexe*. Cela indique que l'impôt augmente non seulement en fonction du revenu, mais également de manière de plus en plus rapide.

```
import matplotlib.pyplot as plt
import numpy as np

# tranche = [10084, 25710, 73516, 158122] # 2020
# tranche = [10225, 26070, 74545, 160336] # 2021
# tranche = [10777, 27478, 78570, 161994] # 2022
# tranche = [10294, 28797, 82341, 177105] # 2023
tranche = [11497, 29315, 83823, 180294]  # 2024
taux = [0.11, 0.30, 0.41, 0.45]


#def Taux(R):
#    if R <= tranche[0]:   return 0
#    elif R <= tranche[1]: return taux[0]
#    elif R <= tranche[2]: return taux[1]
#    elif R <= tranche[3]: return taux[2]
#    else:                 return taux[3]

def Taux(R):
    for t, r in zip([0]+taux, tranche+[float("inf")]):
        if R <= r:
            return t

# next renvoie le premier taux dont la condition est vérifiée, si aucune condition n'est satisfaite, on
↪  retourne taux[-1] (dernier taux)
# Taux = lambda R: next((t for r, t in zip(tranche, [0]+taux) if R <= r), taux[-1])

#def Impot(R):
#    if R <= tranche[0]:
#        return 0
#    elif R <= tranche[1]:
#        return 0 + taux[0] * (R - tranche[0])
#    elif R <= tranche[2]:
#        return 0 + taux[0] * (tranche[1] - tranche[0]) + taux[1] * (R - tranche[1])
#    elif R <= tranche[3]:
#        return (
#            0
#            + taux[0] * (tranche[1] - tranche[0])
#            + taux[1] * (tranche[2] - tranche[1])
#            + taux[2] * (R - tranche[2])
#        )
#    else:
#        return (
#            0
#            + taux[0] * (tranche[1] - tranche[0])
#            + taux[1] * (tranche[2] - tranche[1])
#            + taux[2] * (tranche[3] - tranche[2])
#            + taux[3] * (R - tranche[3])
#        )

def Impot(R):
    imp = 0
    prev = tranche[0]
    for t, nxt in zip(taux, tranche[1:] + [float("inf")]):
        if R > prev:
            imp += t * (min(R, nxt) - prev)
        else:
            break
        prev = nxt
    return imp
```

```
Taux_Reel = lambda R: Impot(R) / R if R > 0 else 0

Reste = lambda R: R - Impot(R)


# ==================================================================
# TESTS
# ==================================================================
R_30 = 30000
print(f"""Pour un célibataire dont le revenu net imposable est de {R_30} €,
sans aucune réduction ni déduction,
son impôt brut est de {Impot(R_30):.2f},
le taux réel d'imposition est {Taux_Reel(R_30)*100:.2f}%
Il lui reste {Reste(R_30):.2f} €. """)

R_100 = 100000
print(f"""\nPour un célibataire dont le revenu net imposable est de {R_100} €,
sans aucune réduction ni déduction,
son impôt brut est de {Impot(R_100):.2f},
le taux réel d'imposition est {Taux_Reel(R_100)*100:.2f}%
Il lui reste {Reste(R_100):.2f} €. """)

# ==================================================================
# COURBES
# ==================================================================
fig = plt.figure(figsize=(20, 12))
plt.rcParams["font.size"] = "20"

# exemple
R_ex = 100000
T_ex = Taux(R_ex)
I_ex = Impot(R_ex)
Reste_ex = Reste(R_ex)
Reel_ex = Taux_Reel(R_ex)

# ==================================================================
xx_ticks = [0] + [val for t in tranche for val in (t, t)] + [180000]
# ==================================================================
rr = np.linspace(0,180000,1001);
# ==================================================================
plt.subplot(2, 2, 1)
yy_tranches = [Taux(x) for x in rr]
plt.plot(rr, yy_tranches, '.')
plt.scatter( R_ex,taux[2], c="purple", marker="x", s=80 )
plt.annotate( (R_ex,taux[2]), (R_ex,taux[2]-0.05), ha='center')
plt.xticks(tranche, rotation=45)
plt.yticks(taux)
plt.xlabel("Revenue")
plt.title("Taux d'imposition marginal")
plt.grid(True)

# ==================================================================
plt.subplot(2, 2, 2)
yy_impot = [Impot(r) for r in rr]
plt.plot(rr, yy_impot, lw=2)
plt.scatter( R_ex,I_ex, c="purple", marker="x", s=80 )
plt.annotate( (R_ex,round(I_ex)), (R_ex,I_ex-0.05), ha='center')
plt.xticks(tranche, rotation=45)
yy_impot_ticks = [Impot(r) for r in xx_ticks]
plt.yticks(yy_impot_ticks)
plt.xlabel("Revenues")
plt.title("Impot à payer")
plt.grid(True)
```

```
# ==================================================================
plt.subplot(2, 2, 3)
rr_fin = np.linspace(0,180000,20001);
yy_reel = [Taux_Reel(r) for r in rr_fin]
plt.plot(rr_fin,yy_reel,lw=2)
plt.scatter( R_ex,Reel_ex, c="purple", marker="x", s=80 )
plt.annotate( (R_ex,round(Reel_ex,2)), (R_ex,Reel_ex-0.05), ha='center')
plt.xticks(tranche, rotation=45)
yy_taux_reel_ticks = [round(Taux_Reel(r),2) for r in xx_ticks]
plt.yticks(yy_taux_reel_ticks,yy_taux_reel_ticks)
plt.xlabel("Revenues")
plt.title("Taux réel")
plt.grid(True)

# ==================================================================
plt.subplot(2, 2, 4)
yy_reste = [Reste(r) for r in rr]
plt.plot(rr, yy_reste, lw=2)
plt.scatter( R_ex,Reste_ex, c="purple", marker="x", s=80 )
plt.annotate( (R_ex,round(Reste_ex,2)), (R_ex,Reste_ex-0.05), ha='center')
plt.plot([0, 180000], [0, 180000], "r:")
plt.xticks(tranche, rotation=45)
yy_reste_ticks = [Reste(r) for r in xx_ticks]
plt.yticks(yy_reste_ticks)
plt.xlabel("Revenues")
plt.title("Ce qui reste après avoir payé l'impôt")
plt.grid(True)

# ==================================================================
plt.tight_layout()
plt.savefig("Images/global.png")
plt.show();
```

```
Pour un célibataire dont le revenu net imposable est de 30000 €,
sans aucune réduction ni déduction,
son impôt brut est de 2165.48,
le taux réel d'imposition est 7.22%
Il lui reste 27834.52 €.

Pour un célibataire dont le revenu net imposable est de 100000 €,
sans aucune réduction ni déduction,
son impôt brut est de 24944.95,
le taux réel d'imposition est 24.94%
Il lui reste 75055.05 €.
```

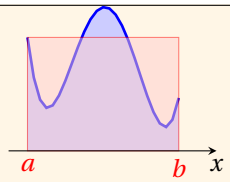
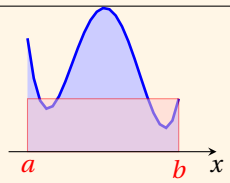
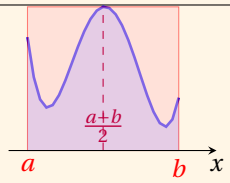
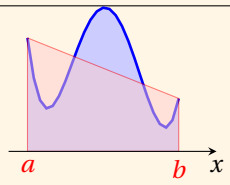
Exercice Bonus 8.27 (Approximations d'intégrales)

Soit f une fonction réelle intégrable sur l'intervalle $[a,b]$. Le calcul explicite de l'intégrale définie

$$\mathrm{I}(f,a,b) \overset{\text{déf}}{=} \int_a^b f(x)\,\mathrm{d}x$$

peut être difficile, voire impossible.

On appelle *formule de quadrature* ou *formule d'intégration numérique* toute formule permettant de calculer une approximation de $\mathrm{I}(f,a,b)$. Dans cet exercice, nous explorons plusieurs méthodes pour approximer l'intégrale d'une fonction f continue sur $[a,b]$:

Rectangle à gauche	$\mathrm{I}(f,a,b) \approx (b-a)f(a)$	
Rectangle à droite	$\mathrm{I}(f,a,b) \approx (b-a)f(b)$	
Point milieu	$\mathrm{I}(f,a,b) \approx (b-a)f\left(\frac{a+b}{2}\right)$	
Trapèze	$\mathrm{I}(f,a,b) \approx \frac{(b-a)}{2}(f(a)+f(b))$	

Pour améliorer la précision, on peut décomposer l'intervalle $[a,b]$ en m sous-intervalles de largeur $\mathrm{H} = \frac{b-a}{m}$, avec $m \geq 1$, et appliquer la formule à chaque sous intervalle en exploitant l'additivité des intégrales. En introduisant les nœuds $x_k = a + k\mathrm{H}$, pour $k = 0, \ldots, m$, cela donne

$$\mathrm{I}(f,a,b) = \sum_{k=0}^{m} \mathrm{I}(f, x_k, x_{k+1}).$$

- **Composite rectangle à gauche**: $\int_a^b f(x)\,\mathrm{d}x \approx \mathrm{H}\sum_{k=0}^{m-1} f(x_k) = \mathrm{H}\sum_{k=0}^{m-1} f(a+k\mathrm{H})$.
- **Composite rectangle à droite**: $\int_a^b f(x)\,\mathrm{d}x \approx \mathrm{H}\sum_{k=1}^{m} f(x_k) = \mathrm{H}\sum_{k=1}^{m} f(a+k\mathrm{H})$.
- **Composite point milieu**: $\int_a^b f(x)\,\mathrm{d}x \approx \mathrm{H}\sum_{k=0}^{m-1} f\left(a+\frac{\mathrm{H}}{2}+k\mathrm{H}\right)$.
- **Composite trapèze**: $\int_a^b f(x)\,\mathrm{d}x \approx \frac{\mathrm{H}}{2}\left(f(a)+2\sum_{k=1}^{m-1} f(a+k\mathrm{H})+f(b)\right)$.

1. **Formules de base**: codez ces méthodes en Python et testez-les sur un exemple simple pour lequel on connaît la solution exacte.
2. **Formules composites**: discrétisez l'intervalle $[a,b]$ en $\mathrm{N}+1$ points avec un pas $\mathrm{H} = \frac{b-a}{\mathrm{N}}$; implémentez les méthodes composites.
3. **Ordre de convergence**:
   - Tracez $\ln(\mathrm{H})$ en fonction de $\ln(\text{erreur})$ pour chaque méthode.
   - Estimez la pente de cette courbe à l'aide de `polyfit` de `numpy`.

Correction

```
rectangle_gauche = lambda f, a, b: f(a) * (b - a)
rectangle_droite = lambda f, a, b: f(b) * (b - a)
rectangle_centre = lambda f, a, b: f((a + b) / 2) * (b - a)
trapeze = lambda f, a, b: ((f(a) + f(b)) / 2) * (b - a)

# Test sur un exemple
import numpy as np
f = lambda x: np.cos(x)  # Exemple de fonction
primitive_f = lambda x: np.sin(x) # const = 0
integral_exacte = lambda a, b: primitive_f(b)-primitive_f(a)  # Integrale exacte

a, b = 0, 1
exact = integral_exacte(a, b)
print("Valeur exacte:", exact)
```

```
# Approximations simples
print("Rectangle gauche:", rectangle_gauche(f, a, b))
print("Rectangle droite:", rectangle_droite(f, a, b))
print("Rectangle centre:", rectangle_centre(f, a, b))
print("Trapeze:", trapeze(f, a, b))
```

```
Valeur exacte: 0.8414709848078965
Rectangle gauche: 1.0
Rectangle droite: 0.5403023058681398
Rectangle centre: 0.8775825618903728
Trapeze: 0.7701511529340699
```

Formules composites

```
def composite_rectangle_gauche(f, a, b, N):
    H = (b - a) / N
    somme = sum(f(a + i * H) for i in range(N))
    return H * somme

def composite_rectangle_droite(f, a, b, N):
    H = (b - a) / N
    somme = sum(f(a + (i + 1) * H) for i in range(N))
    return H * somme

def composite_rectangle_centre(f, a, b, N):
    H = (b - a) / N
    somme = sum(f(a + (i + 0.5) * H) for i in range(N))
    return H * somme

def composite_trapeze(f, a, b, N):
    H = (b - a) / N
    somme = sum(f(a + i * H) for i in range(1, N))
    return H * (0.5 * f(a) + somme + 0.5 * f(b))

# Tests des formules composites
N = 10
print("\nFormules composites (N=10):")
print("Rectangle gauche:", composite_rectangle_gauche(f, a, b, N))
print("Rectangle droite:", composite_rectangle_droite(f, a, b, N))
print("Rectangle centre:", composite_rectangle_centre(f, a, b, N))
print("Trapeze:", composite_trapeze(f, a, b, N))
```

```
Formules composites (N=10):
Rectangle gauche: 0.8637545267950127
Rectangle droite: 0.8177847573818268
Rectangle centre: 0.8418217000072957
Trapeze: 0.8407696420884196
```

Etude de la convergence

```
import matplotlib.pyplot as plt

schemas = {
    "Rectangle gauche": composite_rectangle_gauche,
    "Rectangle droite": composite_rectangle_droite,
    "Rectangle centre": composite_rectangle_centre,
    "Trapeze": composite_trapeze
}

print("\nOrdres de convergence:")
for nom, schema in schemas.items():
    erreurs = []
    H_vals = []
    for N in [10, 20, 40, 80, 160]:
```

```
        H = (b - a) / N
        approximation = schema(f, a, b, N)
        erreur = abs(approximation - exact)
        erreurs.append(erreur)
        H_vals.append(H)

    # Trace de ln(H) et ln(err)
    ln_H = np.log(H_vals)
    ln_err = np.log(erreurs)
    p = np.polyfit(ln_H, ln_err, 1)[0]

    print(f"Ordre de convergence estime pour {nom}: {p}")

    plt.plot(ln_H, ln_err, 'o-', label=f"{nom} (ordre: {p:.2f})")

plt.xlabel("ln(H)")
plt.ylabel("ln(erreur)")
plt.legend()
plt.grid()
plt.title("Convergence des schemas composites")
plt.show()
```

```
Ordres de convergence:
Ordre de convergence estime pour Rectangle gauche: 0.989942807775919
Ordre de convergence estime pour Rectangle droite: 1.0097579919413777
Ordre de convergence estime pour Rectangle centre: 2.000093700796787
Ordre de convergence estime pour Trapeze: 2.0000535452049295
```

ANNEXE A

Les «mauvaises» propriétés des nombres flottants et la notion de précision

Le calcul sur ordinateur repose généralement sur l'utilisation de nombres en virgule flottante, ce qui le distingue du calcul symbolique. Ces nombres sont une manière courante de représenter numériquement des valeurs réelles dans divers langages de programmation, dont Python. Cependant, ils présentent des limites et des caractéristiques indésirables.

Parmi ces caractéristiques, on trouve les débordements, les sous-débordements et la perte de précision. Un débordement se produit quand un nombre flottant devient trop grand pour être correctement représenté en mémoire, souvent lors d'opérations impliquant des valeurs extrêmement grandes. Inversement, un sous-débordement survient lorsqu'un nombre flottant devient trop petit pour être représenté en mémoire, généralement lors de divisions. Quant à la perte de précision, elle se manifeste lorsque les erreurs d'arrondi liées à une opération deviennent supérieures à la précision du nombre flottant.

Les propriétés spécifiques des flottants en Python, notamment leur représentation en double précision sur 64 bits, influent sur la manière dont ces limitations se manifestent. Un nombre flottant est constitué d'un signe, d'une mantisse (aussi appelée "chiffres significatifs") et d'un exposant : $(-1)^{\text{signe}} + 1.\text{mantisse} \times 2^{\text{exposant}}$. En double précision, les flottants Python se composent d'un bit pour le signe, de 52 bits pour la mantisse (ce qui équivaut à environ 15 à 16 décimales significatives) et de 11 bits pour l'exposant, leur permettant de représenter des nombres dans une plage allant de 10^{-308} à 10^{308}.

EXEMPLE (UN CALCUL DE π)

On veut utiliser la propriété mathématique

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

pour calculer la valeur de π. Comme ce calcul est réalisé avec des nombres flottants, il y a plusieurs problèmes de précision :

1. le meilleur calcul de π possible ne pourra donner qu'un arrondi à $\approx 10^{-15}$ près tandis que la vraie valeur de π n'est pas un flottant représentable (il n'est même pas rationnel) ;
2. en utilisant des nombres flottants, chaque opération (/, +, ...) peut faire une erreur d'arrondi (erreur relative de 10^{-15}). Les erreurs peuvent se cumuler.
3. La formule mathématique est infinie. Le calcul informatique sera forcé de s'arrêter après un nombre fini d'itérations.

```
print(4*sum([(-1)**n/(2*n+1) for n in range(1)]))
print(4*sum([(-1)**n/(2*n+1) for n in range(10)]))
print(4*sum([(-1)**n/(2*n+1) for n in range(100)]))
print(4*sum([(-1)**n/(2*n+1) for n in range(1000)]))
```

```
4.0
3.0418396189294024
3.131592903558553
3.1405926538397932
```

Valeur prédéfinie par le module math

```
from math import *
print(pi)
```

```
3.141592653589793
```

A.1. Quelques remarques sur les nombres en Virgule Flottante

En informatique, la notation en virgule flottante (visualisée par un point décimal) permet de représenter une **approximation** d'un nombre réel. Par exemple, 2 est un entier ($2 \in \mathbb{Z}$), tandis que 2.0 est un nombre réel ($2.0 \in \mathbb{R}$). Dans l'ordinateur, ces deux nombres sont deux objets différents et sont stockés différemment :

```
>>> type(2)
<class 'int'>
```

```
>>> type(2.0)
<class 'float'>
```

Un nombre réel est stocké en mémoire selon une représentation spécifique appelée **notation en virgule flottante** sous la forme :

$$\text{significande} \times \text{base}^{\text{exposant}}.$$

Par exemple, la valeur de π est 3.1415926535897… dont l'écriture décimale est infinie (et non périodique). Si l'on suppose que nous avons seulement quatre chiffres significatifs pour une opération, alors π n'est stocké que sous une forme tronquée qui en est une approximation, par exemple 3.1415. Ce nombre peut être alors réécrit (et stocké) comme :

$$3.1415 = 31415 \times 10^{-4}$$

où 31415 est la **significande**, 10 est la **base**, −4 est l'**exposant**.

Comment Stocker un Nombre en Virgule Flottante Les ordinateurs stockent les nombres en virgule flottante selon le modèle suivant :

$$\pm d_1 d_2 ... d_s \times \beta^e$$

où

- β est la base (nous utilisons β = 10 masi dans un ordinateur c'est typiquement β = 2)
- d_i sont des chiffres compris entre 0 et β − 1 avec $d_1 \neq 0$,
- $m \leq e \leq M$, avec m et M appartenant à $\mathbb{Z}$.

Le terme *flottante* vient du fait que la position du point décimal peut varier grâce à l'exposant, comme en notation scientifique.

Un nombre flottant est donc composé de trois parties :

- **Signe** (±)
- **Mantisse** ($d_1 d_2 ... d_s$)
- **Exposant** (β)

Ces différentes parties sont stockées séparément en mémoire et utilisent un nombre de bits défini par la norme **IEEE 754** [1]

Contrairement aux entiers qui ont une précision illimitée, les nombres flottants sont généralement stockés en double précision. Ce compromis permet d'équilibrer *précision* et *intervalle de représentation*. Si vous utilisez plus de bits pour stocker une valeur, vous obtenez une meilleure approximation du nombre réel, mais vous restez toujours limité à l'utilisation de valeurs approchées plutôt que de valeurs exactes.

On peut obtenir les informations sur la précision et la représentation interne des nombres flottants d'une machine avec Python :

```
>>> import sys  # Importation du module sys
>>> sys.float_info  # Obtenir les infos sur les nombres flottants
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738585072014e-308,
↪ min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2,
↪ rounds=1)
```

D'après `sys.float_info`, le plus grand nombre en virgule flottante est

$$\max = 1.7976931348623157e + 308 \quad (1.7976931348623157 \times 10^{308})$$

Les nombres supérieurs au *maximum flottant* sont remplacés par l'infini positif. De même, les nombres inférieurs à un seuil appelé *minimum flottant (normalisé)* sont arrondis à zéro ou traités avec une précision réduite (nombres flottant *dénormalisés*). De plus, la précision des nombres flottants est relative et dépend de leur ordre de grandeur.

Bornes des flottants en Python. Les plus petits et les plus grands nombres flottants normalisés utilisables peuvent être obtenus ainsi :

```
>>> import sys
>>> min_float = sys.float_info.min
>>> max_float = sys.float_info.max
>>> print("Flottant minimum :", min_float)
Flottant minimum : 2.2250738585072014e-308
>>> print("Flottant maximum :", max_float)
Flottant maximum : 1.7976931348623157e+308
```

1. *Simple précision* : 4 octets = 32 bits ; *Double précision* : 8 octets = 64 bits ; *Précision étendue* : 80 bits ; *Quadruple précision* : 16 octets = 128 bits.

Petits nombres : dénormalisés. Les flottants permettent de représenter des nombres plus petits que le *minimum flottant normalisé* en les traitant comme des nombres *dénormalisés*. Ces nombres ont une précision réduite mais restent représentables. Ainsi, 10^{-323} peut encore être représenté (dénormalisé), tandis que 10^{-324} est trop petit et devient 0.0.

```
>>> 10**(-308)  # Correctement représenté, normalisé
1e-308
>>> 10**(-323)  # Correctement représenté, dénormalisé
1e-323
>>> 10**(-324)  # Trop petit, considéré comme nul
0.0
```

Grands nombres. Les nombres dépassant le *maximum flottant* ne peuvent pas être représentés avec exactitude et sont remplacés par une valeur d'infini positif.

Changement de base Un autre problème majeur des calculs en virgule flottante est dû au changement de base. En effet, alors que les nombres réels sont en base 10, les ordinateurs stockent les nombres en base 2, ce qui peut introduire des erreurs d'arrondi. Par exemple, en base 2, le nombre 0.123_2 est défini comme

$$0.123_2 = 1 \times \frac{1}{2} + 2 \times \frac{1}{2^2} + 3 \times \frac{1}{2^3}$$

c'est à dire le nombre $\frac{11}{8} = \frac{11}{8} \times \frac{125}{125} = \frac{1375}{1000} = 1.375_{10}$.

La plupart du temps, cette erreur est négligeable, mais dans certaines applications, elle peut être critique. Il est donc important d'en être conscient et d'utiliser des techniques d'atténuation lorsque nécessaire.

A.1.1. Ce qu'il faut retenir pour travailler avec des nombres flottants

Les flottants diffèrent des entiers en raison de leur représentation et de leur précision limitée. Cela s'explique par deux raisons principales.

- **Représentation binaire des nombres flottants.** Certains nombres décimaux (tels que 0.1) n'ont pas une représentation exacte en binaire. Ils sont donc approchés dès leur stockage, ce qui peut introduire des erreurs dès le début.
- **Nombre fini de chiffres significatifs.** En Python, un flottant sur 64 bits ne peut représenter qu'environ 15 à 16 chiffres significatifs. Les chiffres moins significatifs sont soit tronqués, soit arrondis, ce qui introduit des erreurs d'arrondi.

Ces limitations entraînent plusieurs conséquences :

- **Petits ajouts insignifiants :** Lorsqu'une petite valeur est ajoutée à un très grand nombre, cette modification peut être négligée. Par exemple :

```
>>> x = 1e80  # Un très grand nombre (attention à ne pas utiliser 10**(80), qui est un entier)
>>> y = x + 1
>>> print(y - x)  # Résultat : 0
0.0
```

- **Accumulation d'erreurs :** dans des calculs complexes, les erreurs d'arrondi peuvent s'accumuler, ce qui peut produire des résultats sensiblement différents des valeurs théoriques. Cependant, ces écarts restent négligeables dans la majorité des cas.
- **Comparaison de flottants :** il est déconseillé de vérifier directement l'égalité entre deux flottants. Il est préférable de comparer leur différence en utilisant une tolérance ε : deux nombres a et b sont considérés égaux si $|a - b| < \varepsilon$. Par exemple :

```
>>> a = 0.3 - 0.1
>>> b = 0.2
>>> a == b  # Résultat : False (en raison des erreurs d'arrondi)
False
>>> abs(a - b) < 1.e-14  # Vérification avec ε = 10^-14
True
```

  Plutôt que de choisir soi-même ε, on utilisera de préférence la fonction `isclose` du module `math`, qui réalise exactement cette comparaison :

```
>>> from math import isclose
>>> isclose(a, b)
True
```

Attention (Comparaison de nombres flottants) Lorsqu'on écrit $a = x$, où x est un nombre réel, la valeur a enregistrée en machine est généralement une approximation de x (parfois exacte). Cette approximation est représentée sous forme de nombre flottant (type `float` en Python). En conséquence, bien que les calculs et les comparaisons soient exacts en mathématiques pour des réels, ils ne le sont qu'approximativement dans leur représentation en machine. Ainsi, il est possible que, pour $a = x$ et $b = y$, l'expression `a == b` soit évaluée à `False` alors que x et y sont égaux (et inversement) :

```
>>> 1/10 == 0.1  # réponse attendue : True
True
>>> 1/3 == 0.33333333333333333333  # réponse attendue : False
True
>>> 0.1+0.2 == 0.3  # réponse attendue : True
False
```

Lorsque l'on souhaite tester la valeur d'une variable de type `float`, **il est déconseillé d'utiliser l'opérateur d'égalité**, comme évoqué précédemment (voir la section A.1.1). En effet, les erreurs d'arrondi introduites par la représentation en machine rendent cette approche peu fiable. Pour comparer des nombres flottants, il est préférable de **vérifier si la différence entre eux est inférieure à une certaine tolérance**, choisie en fonction de la précision souhaitée (généralement supérieure à la précision machine). Voici deux exemples équivalents :

```
>>> delta = 1.e-14  # précision
↪  choisie
>>> x = 1/10
>>> y = 0.1
```

```
>>> # Méthode 1 : intervalle
>>> y - delta < x < y + delta
True
```

```
>>> # Méthode 2 : différence
↪  absolue
>>> abs(x - y) < delta
True
```

Dans cet exemple, on teste si x est compris dans l'intervalle $y \pm \delta$. Les deux méthodes sont strictement équivalentes et garantissent une comparaison fiable.

Enfin, pour valider une proximité entre deux valeurs, on peut écrire : [2]

```
>>> assert abs(1.414214 - 2**0.5) < 1e-6
```

Ce test confirme que les deux valeurs sont proches avec une tolérance de 10^{-6}.

Astuce (`math.isclose` : **la comparaison fournie par Python**)
Les méthodes ci-dessus obligent à choisir une tolérance à la main. Le module `math` fournit `isclose`, qui encapsule ce test :

```
>>> from math import isclose
>>> 0.1 + 0.2 == 0.3
False
>>> isclose(0.1 + 0.2, 0.3)
True
```

Par défaut, `isclose(a,b)` considère a et b égaux lorsque

$$|a - b| \leq 10^{-9} \max(|a|, |b|),$$

c'est-à-dire avec une tolérance **relative**, proportionnée à l'ordre de grandeur des nombres comparés. C'est ce qui la rend préférable à un ε **absolu** choisi à l'aveugle : un même ε est trop grossier pour de très petits nombres et trop strict pour de très grands.

La tolérance par défaut est stricte. Sur l'exemple de l'`assert` ci-dessus, où l'on se contentait de 10^{-6}, elle ne suffit pas :

```
>>> isclose(2**0.5, 1.414214)
False
>>> isclose(2**0.5, 1.414214, rel_tol=1e-6)
True
```

Les deux tolérances se règlent séparément, `rel_tol` pour la relative et `abs_tol` pour l'absolue :

2. Le mot clé `assert` est utilisé en Python afin de vérifier que des propositions sont vraies. Ainsi, l'instruction `assert 3 + 5*7 == 38` permet de vérifier que l'expression `3 + 5*7` est bien évaluée à 38. Si c'est le cas, le programme continue de se dérouler normalement. Dans le cas contraire, le programme est interrompu et une erreur est signalée.

```
>>> isclose(1e-20, 2e-20)                    # tolérance relative par défaut
False
>>> isclose(1e-20, 2e-20, abs_tol=1e-15)     # on autorise un écart absolu
True
>>> isclose(1000.0, 1000.1, rel_tol=1e-3)    # on élargit la tolérance relative
True
```

Noter que `abs_tol` vaut 0 par défaut : pour comparer un flottant à 0, la tolérance relative ne sert à rien et il faut donner `abs_tol` explicitement.

```
>>> isclose(1e-18, 0.0)
False
>>> isclose(1e-18, 0.0, abs_tol=1e-12)
True
```

A.2. Ne jamais faire confiance aveuglément aux résultats d'un calcul obtenu avec un ordinateur...

L'expérimentation numérique dans les sciences est un sujet passionnant et un outil incontournable pour de nombreux scientifiques. Malgré la puissance de calcul vertigineuse de nos ordinateurs modernes, et encore plus de certains centres de calculs, il ne faut pas oublier complètement la théorie et prendre à la légère le fonctionnement des machines, sous peine d'avoir quelques surprises...

Observons des calculs quelque peu surprenants.

- Dans le calcul suivant, la différence entre 4.9 et 4.845 devrait être 0.055 mais python nous dit que non :
  ```
  >>> 4.9 - 4.845 == 0.055
  False
  ```
  Pourquoi cela se produit-il ? Si nous regardons $4.9 - 4.845$, nous pouvons voir que nous obtenons en fait
  ```
  >>> 4.9-4.845
  0.055000000000000604
  ```
- Un autre exemple est le suivant qui montre que $0.1 + 0.2 + 0.3 \neq 0.6$:
  ```
  >>> 0.1 + 0.2 + 0.3
  0.6000000000000001
  >>> 0.1 + 0.2 + 0.3 == 0.6
  False
  >>> round(0.1 + 0.2 + 0.3, 5)  == round(0.6, 5)
  True
  ```
- ```
  >>> 0.1 + 0.1 + 0.1 - 0.3
  5.551115123125783e-17
  >>> 1.1 + 2.2
  3.3000000000000003
  ```

Que s'est-il passé ? Tout simplement, les calculs effectués ne sont pas exacts et sont entachés d'erreurs d'arrondis. En effet, tout nombre réel possède un développement décimal soit fini soit illimité. Parmi les nombres réels, on peut alors distinguer les rationnels (dont le développement décimal est soit fini soit illimité et périodique à partir d'un certain rang) des irrationnels (dont le développement décimal est illimité et non périodique). Il est aisé de concevoir qu'il n'est pas possible pour un ordinateur de représenter de manière exacte un développement décimal illimité, mais même la représentation des développements décimaux finis n'est pas toujours possible. En effet, un ordinateur stocke les nombres non pas en base 10 mais en base 2. Or, un nombre rationnel peut tout à fait posséder un développement décimal fini et un développement binaire illimité ! C'est le cas des décimaux 1.1 et 2.2 qui, en base 2, s'écrivent $01.\overline{01}$ et $10.\overline{001}$ respectivement.

Remarque 18

Un exemple fréquent d'arrondi se produit avec la simple instruction `t=0.1` La valeur mathématique t stockée dans `t` n'est pas exactement 0.1 car l'expression de la fraction décimale $1/10$ en binaire nécessite une série infinie. En effet,

$$\frac{1}{10} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{0}{2^{10}} + \frac{0}{2^{11}} + \cdots$$

La séquence de coefficients 1, 1, 0, 0 se répète à l'infini. En regroupant les termes résultants quatre par quatre, on exprime $1/10$ comme

$$\frac{1}{10} = \frac{1}{2^4}\left(1 + \sum_{i=1}^{\infty} \frac{9}{16^i}\right).$$
  ```

Les nombres à virgule flottante de part et d'autre de 1/10 sont obtenus en terminant la partie fractionnaire de cette série après 52 termes binaires ou 13 termes hexadécimaux et en arrondissant le dernier terme vers le haut ou vers le bas. Ainsi,

$$t_1 < t < t_2 \quad \text{où} \quad t_1 = \frac{1}{2^4}\left(1+\sum_{i=1}^{12}\frac{9}{16^i}+\frac{9}{16^{13}}\right), \quad t_2 = \frac{1}{2^4}\left(1+\sum_{i=1}^{12}\frac{9}{16^i}+\frac{10}{16^{13}}\right).$$

Il se trouve que 1/10 est plus proche de t_2 que t_1 ainsi `t`$= t_2$.

En résumé, la valeur stockée dans `t` est très proche de, mais pas exactement égale à, 0.1. La distinction est parfois importante. Par exemple, la quantité 0.3/0.1 n'est pas exactement égale à 3 car le numérateur réel est un peu inférieur à 0.3 et le dénominateur réel est un peu plus grand que 0.1 :

```
>>> 0.3/0.1
2.9999999999999996
```

Source : Cleve's Corner : Cleve Moler on Mathematics and Computing https://blogs.mathworks.com/cleve/2014/07/07/floating-point-numbers/

Voici quelque source d'erreurs :

Représentation décimale inexacte Dans l'exemple ci-dessous 1.2 n'est pas représentable en machine. L'ordinateur utilise «le flottant représentable le plus proche de 1.2»

```
>>> 1.2 - 1 - 0.2
-5.551115123125783e-17
```

Non-commutativité

```
>>> 1 + 1e-16 - 1
0.0
>>> -1 + 1e-16 + 1
1.1102230246251565e-16
```

Erreurs d'arrondis

```
>>> 1/3 - 1/4 - 1/12
-1.3877787807814457e-17
```

Conséquences

- On ne peut pas espérer de résultat exact
- La précision du calcul dépend de beaucoup d'éléments
- En général, pour éviter les pertes de précision, on essayera autant que faire se peut d'éviter :
  - de soustraire deux nombres très proches
  - d'additionner ou de soustraire deux nombres d'ordres de grandeur très différents. Ainsi, pour calculer une somme de termes ayant des ordres de grandeur très différents (par exemple dans le calcul des sommes partielles d'une série), on appliquera le principe dit "de la photo de classe" : les petits devant, les grands derrière.
- **tester** `x == flottant` **est presque toujours une erreur**, on utilisera plutôt `math.isclose(x,flottant)`, ou à défaut `abs(x-flottant)<1.e-10` par exemple.
- "Si on s'y prend bien", on perd $\approx 10^{-16}$ en précision relative à chaque calcul $\Rightarrow$ acceptable par rapport à la précision des données.
- "Si on s'y prend mal", le résultat peut être complètement faux !

Illustrons ce problème d'arrondis en partant de l'identité suivante :

$$xy = \left(\frac{x+y}{2}\right)^2 - \left(\frac{x-y}{2}\right)^2.$$

Dans le programme suivant on compare les deux membres de cette égalité pour des nombres x et y de plus en plus grands :

```
prod = lambda x,y : x*y
diff = lambda x,y : ((x+y)/2)**2-((x-y)/2)**2

a = 6553.99
b = a+1

from tabulate import tabulate
```

```
T = []
T.append(["a","b","prod(a,b)-diff(a,b)"])
for i in range(6):
    produit = prod(a,b)
    difference = diff(a,b)
    T.append( [a,b,produit-difference] )
    a, b = produit, a+1

print(tabulate(T, headers="firstrow",floatfmt="1.10e"))
```

On constate que la divergence est spectaculaire :

```
                a                 b    prod(a,b)-diff(a,b)
----------------  ----------------  ---------------------
6.5539900000e+03  6.5549900000e+03       0.0000000000e+00
4.2961338910e+07  6.5549900000e+03      -5.8593750000e-02
2.8161114694e+11  4.2961339910e+07       1.6670720000e+06
1.2098392206e+19  2.8161114694e+11      -4.7982566402e+21
3.4070421054e+30  1.2098392206e+19       1.0466488703e+44
4.1219731654e+49  3.4070421054e+30       1.4043736132e+80
```

Le module `fractions` Pour corriger ce problème on peut utiliser le module `fractions` :

```
from fractions import Fraction
T = []
print(f'{"Float":*^20} vs {"Fraction":*^19}')
T.append([0.1 + 0.1 + 0.1 - 0.3, Fraction(1,10) + Fraction(1,10) + Fraction(1,10) - Fraction(3,10)])
T.append([1.1 + 2.2, Fraction(11,10) + Fraction(22,10)])
T.append([1.0 / 3 - 1.0 / 4 - 1.0 / 12, Fraction(1,3) + Fraction(1,4) - Fraction(1,12)])
T.append([ 1 + 1e-16 - 1, Fraction(1,1) + Fraction(1,10**16) - Fraction(1,1)])
T.append([-1 + 1e-16 + 1, -Fraction(1,1) + Fraction(1,10**16) + Fraction(1,1)])
T.append([1.2 - 1.0 - 0.2, Fraction(6,5) - Fraction(1,1) - Fraction(1,5)])
for t in T:
    print(f"{t[0]:<+.17f} vs {t[1]}")
```

```
*******Float******** vs *****Fraction******
+0.00000000000000006 vs 0
+3.30000000000000027 vs 33/10
-0.00000000000000001 vs 1/2
+0.00000000000000000 vs 1/10000000000000000
+0.00000000000000011 vs 1/10000000000000000
-0.00000000000000006 vs 0
```

Revenons sur notre exemple :

```
from fractions import Fraction
from tabulate import tabulate
T = []

prod = lambda x,y : x*y
diff = lambda x,y : Fraction(x+y,2)**2-Fraction(x-y,2)**2

a = Fraction(655399,100)
b = a+1

T.append(["a","b","prod(a,b)-diff(a,b)"])
for i in range(6):
    produit = prod(a,b)
    difference = diff(a,b)
    T.append( [a,b,produit-difference] )
    a, b = produit, a+1

print(tabulate(T, headers="firstrow",floatfmt="1.15e"))
```

```
                    a                     b    prod(a,b)-diff(a,b)
---------------------  ---------------------  ---------------------
```

```
6.553990000000000e+03  6.554990000000000e+03  0.000000000000000e+00
4.296133891010000e+07  6.554990000000000e+03  0.000000000000000e+00
2.816111469423164e+11  4.296133991010000e+07  0.000000000000000e+00
1.209839220626197e+19  2.816111469433164e+11  0.000000000000000e+00
3.407042105375514e+30  1.209839220626197e+19  0.000000000000000e+00
4.121973165408151e+49  3.407042105375514e+30  0.000000000000000e+00
```

Remarque 19

Voici deux exemples de désastres causés par une mauvaise gestion des erreurs d'arrondi :

- Le 25 février 1991, pendant la Guerre du Golfe, une batterie américaine de missiles Patriot située à Dharan en Arabie Saoudite a manqué l'interception d'un missile Scud irakien, qui a frappé un baraquement de l'armée américaine, causant la mort de 28 soldats. Une commission d'enquête a déterminé que l'échec était dû à un calcul incorrect du temps de parcours du missile Scud, causé par un problème d'arrondi lié à la représentation des nombres en virgule fixe sur 24 bits utilisée par l'ordinateur de bord du système Patriot. Le temps était compté en dixièmes de seconde par l'horloge interne du système, mais la représentation binaire de $1/10$ n'a pas d'écriture finie en binaire : $1/10 = 0.1$ (dans le système décimal) $= 0.0001100110011001100110011\ldots$ (dans le système binaire). L'ordinateur de bord arrondissait $1/10$ à 24 chiffres, ce qui a entraîné une petite erreur dans le décompte du temps pour chaque dixième de seconde. Environ 100 heures s'étaient écoulées depuis l'allumage du système Patriot, ce qui avait entraîné une accumulation des erreurs d'arrondi de 0,34 s. Pendant ce temps, le missile Scud avait parcouru environ 500 m, expliquant ainsi pourquoi le Patriot avait manqué sa cible.
- Le 4 juin 1996, le lancement d'une fusée Ariane 5 a connu un accident et a explosé 40 secondes après son décollage. Le coût total de la fusée et de son chargement était estimé à 500 millions de dollars. Après deux semaines, la commission d'enquête a conclu que l'incident était dû à une erreur de programmation dans le système inertiel de référence. Au cours de la conversion d'un nombre codé en virgule flottante sur 64 bits (représentant la vitesse horizontale de la fusée par rapport à la plate-forme de lancement), celui-ci a été converti en un entier sur 16 bits. Malheureusement, le nombre en question était supérieur à 32 768, la valeur maximale pouvant être codée sur 16 bits, ce qui a conduit à une erreur de conversion.

A.3. Exercices

Exercice A.1 (Devine le résultat)
Expliquer les résultats suivants :

```
>>> print(0.2+0.3+0.4)
0.9
>>> print(0.3+0.4+0.2)
0.8999999999999999
>>> print(0.4+0.2+0.3)
0.9000000000000001
>>> print(0.2+0.4)
0.6000000000000001
>>> print(  (-7.35e22 + 7.35e22) + 100.0  )
100.0
>>> print(  -7.35e22 + (7.35e22 + 100.0)  )
0.0
```

Correction
Python effectue les opérations de gauche à droite. Pour la première expression, il calcule déjà `0.2 + 0.3`, puis ajoute au résultat `0.4`. Pour la seconde opération, il calcule `0.3 + 0.4`, puis ajoute au résultat `0.2`. Or ces nombres n'ont pas une écriture finie en base 2 et sont donc approchés. Les erreurs sur le dernier chiffre provoquent ces différences.

Le module `fractions` permet de faire des calculs exacts sur les rationnels :

```
>>> from fractions import Fraction
>>> print(Fraction(2, 10) + Fraction(3, 10) + Fraction(4, 10))
9/10
```

Pour les deux derniers exemples, le problème est liée à la non associativité des calculs en virgule flottante.

Exercice A.2 (Qui est plus grand ?)
Parmi A et B, qui est plus grand si

$$A = \frac{2^{2021}-1}{4^{2021}-2^{2021}+1}, \qquad B = \frac{2^{2021}+1}{4^{2021}+2^{2021}+1}?$$

Correction
Naïvement on pourrait essayer ceci :

```
>>> A = (2**2021-1)/(4**2021-2**2021+1)
>>> B = (2**2021+1)/(4**2021+2**2021+1)
>>> print(A-B)
0.0
```

et en déduire que $A = B$, mais regardons un peu mieux :

```
>>> A = (2**2021-1)/(4**2021-2**2021+1)
>>> B = (2**2021+1)/(4**2021+2**2021+1)
>>> print(f"{A=}, {B=}")
A=0.0, B=0.0
```

Or, ceci ce n'est pas possible car $2^{2021} \pm 1 \neq 0$.

Posons $x = 2^{2021} > 0$ alors $A = \dfrac{x-1}{x^2-x+1}$ et $B = \dfrac{x+1}{x^2+x+1}$ ainsi

$$A - B = \frac{(x-1)(x^2+x+1)-(x+1)(x^2-x+1)}{(x^2-x+1)(x^2+x+1)} = \frac{(x^3+x^2+x-x^2-x-1)-(x^3-x^2+x+x^2-x+1)}{(x^2-x+1)(x^2+x+1)} = \frac{-2}{(x^2-x+1)(x^2+x+1)} < 0$$

Une autre stratégie de calcul est la suivante :

$$\frac{1}{A} - \frac{1}{B} = \frac{x(x-1)+1}{x-1} - \frac{x(x+1)+1}{x+1} = x + \frac{1}{x-1} - x - \frac{1}{x+1} = \frac{1}{x-1} - \frac{1}{x+1} > 0.$$

Pour calculer la bonne valeur nous allons utiliser le module `fractions` qui évite les erreurs d'arrondis :

```
from fractions import Fraction
A = Fraction(2**2021-1,4**2021-2**2021+1)
B = Fraction(2**2021+1,4**2021+2**2021+1)
print(A-B)
```

On voit qu'il s'agit d'une fraction avec un numérateur négatif et un dénominateur positif, ainsi A < B.

Exercice A.3 (Équation quadratique)

On se propose de calculer les racines de $3x^2 + 10^9 x + 5 = 0$.

On peut calculer les racines directement selon la formule

$$r_1 = \frac{-b-\sqrt{d}}{2a}, \qquad r_2 = \frac{-b+\sqrt{d}}{2a}, \qquad d = b^2 - 4ac.$$

Sinon, on peut calculer r_1 et en déduire r_2 selon les formules suivantes :

- la produit des racines vaut c donc $r_2 = c/r_1$,
- le somme des racines vaut $-b$ donc $r_2 = -b - r_1$,

Ou encore utiliser la forme conjuguée pour calculer r_2 :

$$r_2 = \frac{\sqrt{d}-b}{2a} = \frac{\sqrt{d}-b}{2a} \times \frac{\sqrt{d}+b}{\sqrt{d}+b} = \frac{d-b^2}{2a(\sqrt{d}+b)},$$

ou une version simplifiée

$$r_2 = \frac{d-b^2}{2a(\sqrt{d}+b)} = \frac{b^2-4ac-b^2}{2a(\sqrt{d}+b)} = \frac{-2c}{\sqrt{d}+b}.$$

Comparer les différentes valeurs obtenues pour r_2.

Correction

```
from math import sqrt
a = 3; b = 1.0e9; c = 5;
d = b**2 - 4*a*c;
s = sqrt(d);
print(f"{b = :.15e}")
print(f"{d = :.15e} La valeur exacte est 10^18-60")
print(f"{s = :.15e}")
r1 = (-b - s)/(2*a);
print(f"{r1 = }")

r2 = (-b + s)/(2*a);
print(f"Formule de base {r2 = }")
r2 = c/r1;
print(f"Formule utilisant le produit {r2 = }")
r2 = -b-r1;
print(f"Formule utilisant la somme {r2 = }")
r2 = (d-b**2)/(2*a*(b+s));
print(f"Formule utilisant la forme conjuguée {r2 = }")
r2 = -2*c/(b+s);
print(f"Formule utilisant la forme simplifiée {r2 = }")
```

```
b = 1.000000000000000e+09
d = 1.000000000000000e+18 La valeur exacte est 10^18-60
s = 1.000000000000000e+09
r1 = -333333333.3333333
Formule de base r2 = 0.0
Formule utilisant le produit r2 = -1.5000000000000002e-08
Formule utilisant la somme r2 = -666666666.6666667
Formule utilisant la forme conjuguée r2 = 0.0
Formule utilisant la forme simplifiée r2 = -5e-09
```

La dernière écriture évite (enfin) la perte de précision car elle évite le problème de la soustraction de nombres presque égaux.

Vérifions avec du calcul formel :

```
import sympy as sp
sp.var("x")
a = 3; b = 10**9; c = 5;
d = b**2 - 4*a*c;
s = sp.sqrt(d);
print(f"{b = }")
print(f"{d = }")
print(f"{s = }")
sol = sp.solve(a*x**2+b*x+c)
print(sol)
print([float(s) for s in sol])
```

```
b = 1000000000
d = 999999999999999940
s = 2*sqrt(249999999999999985)
[-500000000/3 - sqrt(249999999999999985)/3, -500000000/3 + sqrt(249999999999999985)/3]
[-333333333.3333333, -5e-09]
```

Exercice A.4 (Integer VS Floating point number)

Expliquer les résultats suivants :

```
>>> int(12312312312312312312 3.0)
12312312312312312627 2
>>> int(12312312312312312312 3)
12312312312312312312 3

>>> int(12312312312312312312 3.0 % 10**9)
123126272
>>> int(12312312312312312312 3 % 1e9)
123126272
```

Correction

Dans Python, les nombres entiers ont une précision infinie tandis que les nombres floating point ont une précision finie.

Exercice A.5 (Un contre-exemple du dernier théorème de Fermat ?)

Le dernier théorème de Fermat (prouvé par Andriew Wiles en 1994) affirme que, pour tout entier $n > 2$, trois entiers positifs x, y et z ne peuvent pas satisfaire l'équation $x^n + y^n - z^n = 0$. Expliquez le résultat de cette commande qui donne un contre-exemple apparent au théorème :

```
>>> 844487.**5 + 1288439.**5 - 1318202.**5
0.0
```

Correction

Source : https://scipython.com/book/chapter-9-general-scientific-programming/questions/

Le problème, bien sûr, vient de l'utilisation des nombres à virgule flottante double précision et que la différence entre la somme des deux premiers termes et celle du troisième est inférieure à la précision de cette représentation.

Si on utilise des entiers on a bien :

```
>>> 844487**5 + 1288439**5 - 1318202**5
-235305158626
```

En effet, la précision finie de la représentation en virgule flottante utilisée tronque les décimales avant que cette différence soit apparente :

```
>>> print("Exact =", 844487**5 + 1288439**5, "Arrondis =", 844487.**5 + 1288439.**5)
Exact = 398024523518563901305561949740 6 Arrondis = 3.980245235185639e+30
>>> print("Exact =", 1318202**5, "Arrondis =", 1318202.**5)
Exact = 398024523518563901329092465603 2 Arrondis = 3.980245235185639e+30
```

Ceci est un exemple d'annulation catastrophique.

Exercice A.6 (Suites)
Calculer analytiquement et numériquement les premiers 100 termes des suites suivantes :

$$\begin{cases} u_0 = \frac{1}{4}, \\ u_{n+1} = 5u_n - 1, \end{cases} \qquad \begin{cases} v_0 = \frac{1}{5}, \\ v_{n+1} = 6v_n - 1, \end{cases} \qquad \begin{cases} w_0 = \frac{1}{3}, \\ w_{n+1} = 4v_w - 1. \end{cases}$$

Correction
On a clairement $u_i = \frac{1}{4}$, $v_i = \frac{1}{5}$ et $w_i = \frac{1}{3}$ pour tout $i \in \mathbb{N}$. Cependant, lorsqu'on calcule les 100 premiers termes de ces suites avec Python (ou un autre langage de programmation), quelques surprises apparaissent.

Si on écrit

```
n=0
u = 1/4
print(f"u_{n} = {u}")
for n in range(1,30):
    u = 5*u-1
    print(f"u_{n} = {u}")
```

on trouve bien $u_i = 0.25$ pour tout i :

```
u_0 = 0.25          u_6 = 0.25          u_12 = 0.25         u_18 = 0.25         u_24 = 0.25
u_1 = 0.25          u_7 = 0.25          u_13 = 0.25         u_19 = 0.25         u_25 = 0.25
u_2 = 0.25          u_8 = 0.25          u_14 = 0.25         u_20 = 0.25         u_26 = 0.25
u_3 = 0.25          u_9 = 0.25          u_15 = 0.25         u_21 = 0.25         u_27 = 0.25
u_4 = 0.25          u_10 = 0.25         u_16 = 0.25         u_22 = 0.25         u_28 = 0.25
u_5 = 0.25          u_11 = 0.25         u_17 = 0.25         u_23 = 0.25         u_29 = 0.25
```

Mais si on écrit

```
n=0
v = 1/5
print(f"v_{n} = {v}")
for n in range(1,30):
    v = 6*v-1
    print(f"v_{n} = {v}")
```

on obtient $v_i \simeq 0.2$ pour $i = 0, \ldots, 5$, ensuite les erreurs d'arrondis commencent à se voir :

```
v_0 = 0.2                          v_10 = 0.20000000179015842         v_20 = 0.3082440341822803
v_1 = 0.20000000000000018          v_11 = 0.20000001074095053         v_21 = 0.8494642050936818
v_2 = 0.20000000000000107          v_12 = 0.20000006444570317         v_22 = 4.096785230562091
v_3 = 0.2000000000000064           v_13 = 0.20000038667421904         v_23 = 23.580711383372545
v_4 = 0.20000000000003837          v_14 = 0.20000232004531426         v_24 = 140.48426830023527
v_5 = 0.20000000000023022          v_15 = 0.20001392027188558         v_25 = 841.9056098014116
v_6 = 0.2000000000013813           v_16 = 0.2000835216313135          v_26 = 5050.43365880847
v_7 = 0.20000000000828777          v_17 = 0.20050112978788093         v_27 = 30301.60195285082
v_8 = 0.20000000004972662          v_18 = 0.20300677872728556         v_28 = 181808.6117171049
v_9 = 0.20000000029835974          v_19 = 0.21804067236371338         v_29 = 1090850.6703026295
```

De même

```
n=0
w = 1/3
print(f"w_{n} = {w}")
for n in range(1,41):
    w = 4*w-1
    print(f"w_{n} = {w}")
```

À la vingtième répétition, le résultat est $w_{20} = 0.33331298828125$ ce qui est déjà assez éloigné de $1/3$. À la quarantième répétition de la ligne, le résultat est $w_{40} = -22369621.0$ ce qui n'a plus rien à voir. En fait, l'erreur sur l'arrondi se cumule et le résultat devient complètement absurde.

```
w_0 = 0.3333333333333333
w_1 = 0.33333333333333326
w_2 = 0.33333333333333304
w_3 = 0.33333333333333215
w_4 = 0.3333333333333286
w_5 = 0.3333333333333144
w_6 = 0.33333333333325754
w_7 = 0.33333333333303017
w_8 = 0.3333333333321207
w_9 = 0.3333333333284827
w_10 = 0.3333333333139308
w_11 = 0.3333333332557231
w_12 = 0.3333333330228925
w_13 = 0.3333333320915699
w_14 = 0.3333333283662796
w_15 = 0.3333333134651184
w_16 = 0.33333325386047363
w_17 = 0.33333301544189453
w_18 = 0.3333320617675781
w_19 = 0.3333282470703125
w_20 = 0.33331298828125
w_21 = 0.333251953125
w_22 = 0.3330078125
w_23 = 0.33203125
w_24 = 0.328125
w_25 = 0.3125
w_26 = 0.25
w_27 = 0.0
w_28 = -1.0
w_29 = -5.0
w_30 = -21.0
w_31 = -85.0
w_32 = -341.0
w_33 = -1365.0
w_34 = -5461.0
w_35 = -21845.0
w_36 = -87381.0
w_37 = -349525.0
w_38 = -1398101.0
w_39 = -5592405.0
w_40 = -22369621.0
```

Explication : le calcul avec les flottants est un calcul approché. Suivant l'expression utilisée, le résultat peut être différent. Le problème avec le calcul de $4u_n - 1$ est que `4 * 1/3 -1` n'est pas égal en machine à `1/3`:

```
>>> 4 * 1/3 - 1
0.33333333333333326
>>> 1/3
0.3333333333333333
```

Cependant, l'expression de u_{n+1} en fonction de u_n peut s'écrire de différentes manières. Par exemple $u_{n+1} = 4 \times u_n - 1$, ou $u_{n+1} = 4 \times \left(u_n - \frac{1}{4}\right)$, ou encore $u_{n+1} = 3 \times \left(\frac{4}{3}u_n - \frac{1}{3}\right)$ etc. Considérons la dernière expression ; on peut vérifier que `3 * ((4/3)*(1/3) - 1/3)` vaut bien `1/3`:

```
>>> 3 * ((4/3)*(1/3) - 1/3)
0.3333333333333333
>>> 1/3
0.3333333333333333
```

On pourrait donc écrire la solution ainsi :

```
n=0
w = 1/3
print(f"w_{n} = {w}")
for n in range(1,41):
    w = 3*(4*w/3-1/3)
print(f"w_{n} = {w}")
```

```
w_0 = 0.3333333333333333
w_40 = 0.3333333333333333
```

Pour calculer la bonne valeur sans se soucier des erreurs d'arrondis, nous allons utiliser le module `fractions` :

```
from fractions import Fraction
n=0
v = Fraction(1,5)
print(f"v_{n} = {v}")
for n in range(1,30):
    v = 6*v-1
    print(f"v_{n} = {v}")
```

```
v_0 = 1/5
v_1 = 1/5
v_2 = 1/5
v_3 = 1/5
v_4 = 1/5
v_5 = 1/5
v_6 = 1/5
v_7 = 1/5
v_8 = 1/5
v_9 = 1/5
v_10 = 1/5
v_11 = 1/5
v_12 = 1/5
v_13 = 1/5
v_14 = 1/5
v_15 = 1/5
v_16 = 1/5
v_17 = 1/5
v_18 = 1/5
v_19 = 1/5
v_20 = 1/5
v_21 = 1/5
v_22 = 1/5
v_23 = 1/5
v_24 = 1/5
v_25 = 1/5
v_26 = 1/5
v_27 = 1/5
v_28 = 1/5
v_29 = 1/5
```

```
print("♡" * 8)
n=0
w = Fraction(1,3)
print(f"w_{n} = {w}")
for n in range(1,41):
    w = 4*w-1
    print(f"w_{n} = {w}")
```

```
♡♡♡♡♡♡♡♡
w_0 = 1/3
w_1 = 1/3
w_2 = 1/3
w_3 = 1/3
w_4 = 1/3
w_5 = 1/3
w_6 = 1/3
w_7 = 1/3
w_8 = 1/3
w_9 = 1/3
w_10 = 1/3
w_11 = 1/3
w_12 = 1/3
w_13 = 1/3
w_14 = 1/3
w_15 = 1/3
w_16 = 1/3
w_17 = 1/3
w_18 = 1/3
w_19 = 1/3
w_20 = 1/3
w_21 = 1/3
w_22 = 1/3
```

```
w_23 = 1/3      w_26 = 1/3      w_29 = 1/3      w_32 = 1/3      w_35 = 1/3      w_38 = 1/3
w_24 = 1/3      w_27 = 1/3      w_30 = 1/3      w_33 = 1/3      w_36 = 1/3      w_39 = 1/3
w_25 = 1/3      w_28 = 1/3      w_31 = 1/3      w_34 = 1/3      w_37 = 1/3      w_40 = 1/3
```

Exercice A.7 (Suite de Muller)

Considérons la suite

$$\begin{cases} x_0 = 4, \\ x_1 = 4.25, \\ x_{n+1} = 108 - \frac{815 - \frac{1500}{x_{n-1}}}{x_n}. \end{cases}$$

On peut montrer que $\lim_{n\to+\infty} x_n = 5$ (voir par exemple https://scipython.com/blog/mullers-recurrence/) Qu'obtient-on numériquement ?

Correction

```python
x=[4, 4.25]
print(f"x_{0}={x[0]}")
print(f"x_{1}={x[1]}")
for i in range(2,30):
    x.append(108-(815-(1500/x[-2]))/x[-1])
    print(f"x_{i}={x[i]}")
```

```
x_0=4                          x_10=4.987909232795786         x_20=100.00001247862016
x_1=4.25                       x_11=4.991362641314552         x_21=100.00000062392161
x_2=4.470588235294116          x_12=4.967455095552268         x_22=100.0000000311958
x_3=4.6447368421052175         x_13=4.42969049830883          x_23=100.00000000155978
x_4=4.770538243625083          x_14=-7.817236578459315        x_24=100.00000000007799
x_5=4.855700712568563          x_15=168.93916767106458        x_25=100.0000000000039
x_6=4.91084749866063           x_16=102.03996315205927        x_26=100.0000000000002
x_7=4.945537395530508          x_17=100.0999475162497         x_27=100.00000000000001
x_8=4.966962408040999          x_18=100.00499204097244        x_28=100.0
x_9=4.980042204293014          x_19=100.0002495792373         x_29=100.0
```

Pour calculer la bonne valeur nous allons utiliser le module 'fractions' qui évite les erreurs d'arrondis :

```python
from fractions import Fraction
x=[4, Fraction(17, 4)]
print(f"x_{0}={x[0]}")
print(f"x_{1}={x[1]}")
for i in range(2,30):
    x.append(108 - Fraction((815 - Fraction(1500, x[-2])), x[-1]))
    print(f"x_{i}={x[i]} ≈ {float(x[i])}")
```

```
x_0=4
x_1=17/4
x_2=76/17 ≈ 4.470588235294118
x_3=353/76 ≈ 4.644736842105263
x_4=1684/353 ≈ 4.770538243626063
x_5=8177/1684 ≈ 4.855700712589074
x_6=40156/8177 ≈ 4.910847499082793
x_7=198593/40156 ≈ 4.945537404123916
x_8=986404/198593 ≈ 4.966962581762701
x_9=4912337/986404 ≈ 4.980045701355631
x_10=24502636/4912337 ≈ 4.987979448478392
x_11=122336033/24502636 ≈ 4.992770288062068
x_12=611148724/122336033 ≈ 4.995655891506634
x_13=3054149297/611148724 ≈ 4.997391268381344
x_14=15265963516/3054149297 ≈ 4.998433943944817
x_15=76315468673/15265963516 ≈ 4.999060071970894
x_16=381534296644/76315468673 ≈ 4.999435937146839
x_17=1907542343057/381534296644 ≈ 4.999661524103767
x_18=9537324294796/1907542343057 ≈ 4.9997969007134175
```

```
x_19=47685459212513/9537324294796 ≈ 4.999878135477931
x_20=238423809278164/47685459212513 ≈ 4.9999268795046
x_21=1192108586037617/238423809278164 ≈ 4.999956127061158
x_22=5960511549128476/1192108586037617 ≈ 4.9999736760057125
x_23=29802463602463553/5960511549128476 ≈ 4.999984205520272
x_24=149012035582781284/29802463602463553 ≈ 4.999990523282228
x_25=745059330625296977/149012035582781284 ≈ 4.99999431395856
x_26=3725294111260656556/745059330625296977 ≈ 4.999996588371256
x_27=18626462930705797793/3725294111260656556 ≈ 4.9999979530213565
x_28=93132291776736534004/18626462930705797793 ≈ 4.999998771812312
x_29=465661390253305305137/93132291776736534004 ≈ 4.999999263087206
```

Exercice A.8 (Défi Turing n°86 – Le curieux 2 000-ème terme)

Considérons la suite

$$\begin{cases} x_0 = \dfrac{3}{2}, \\ x_1 = \dfrac{5}{2}, \\ x_{n+1} = 2003 - \dfrac{6002}{x_n} + \dfrac{4000}{x_n x_{n-1}}. \end{cases}$$

Que vaut u_{2000} ?

Correction

```python
x=[3/2, 5/2]
print(f"x_{0}={x[0]}")
print(f"x_{1}={x[1]}")
for i in range(2,2001):
    x.append(2003-6002/x[-1]+4000/(x[-1]*x[-2]))
print(f"x_{i}={x[i]}")
```

```
x_0=1.5                    x_1=2.5                    x_2000=2000.0
```

Pour calculer la bonne valeur nous allons utiliser le module ‘fractions‘ qui évite les erreurs d’arrondis :

```python
from fractions import Fraction
x=[Fraction(3,2),Fraction(5,2)]
print(f"x_{0}={x[0]}")
print(f"x_{1}={x[1]}")
for i in range(2,2001):
    x.append(2003-Fraction(6002,x[-1])+Fraction(4000,x[-2]*x[-1]))
print(f"x_{i} ≈ {float(x[i])}")
```

```
x_0=3/2                    x_1=5/2                    x_2000 ≈ 2000.0
```

Exercice A.9 (Algorithme de Brent et Salamin)

Soient $(a_n, b_n, s_n)_{n\in\mathbb{N}}$ trois suites définies par

$$\begin{cases} a_0 = 1, \\ b_0 = \frac{1}{\sqrt{2}}, \\ s_0 = \frac{1}{2}, \\ a_{n+1} = \frac{a_n+b_n}{2}, \\ b_{n+1} = \sqrt{a_n b_n}, \\ s_{n+1} = s_n - 2^{n+1}(a_{n+1}^2 - b_{n+1}^2) \end{cases}$$

Vérifier que $p_n \overset{\text{déf}}{=} \dfrac{2a_n^2}{b_n} \to \pi$.

Correction
On remarque que, après quelques itérations, les erreurs d'arrondis empêchent la convergence. Pour un calcul exacte on peu utiliser le module `sympy` :

```
from math import pi
a, b, s = 1, 1/2**0.5, 1/2
N = 9
for n in range(N):
    a, b = (a+b)/2, (a*b)**0.5
    s = s-2**(n+1)*(a**2-b**2)
    print(f"p_{n}={2*a**2/s:1.15f},
    ↪  |erreur_{n}|={abs(pi-2*a**2/s):g}")
```

```
p_0=3.187672642712109, |erreur_0|=0.04608
p_1=3.141680293297657, |erreur_1|=8.76397e-05
p_2=3.141592653895460, |erreur_2|=3.05667e-10
p_3=3.141592653589831, |erreur_3|=3.81917e-14
p_4=3.141592653589880, |erreur_4|=8.70415e-14
p_5=3.141592653589978, |erreur_5|=1.84741e-13
p_6=3.141592653590173, |erreur_6|=3.8014e-13
p_7=3.141592653590564, |erreur_7|=7.70939e-13
p_8=3.141592653591346, |erreur_8|=1.55254e-12
```

```
import sympy
a, b, s = 1, 1/sympy.sqrt(2), sympy.S(1)/2
N = 6
for i in range(N):
    a, b = (a+b)/2, sympy.sqrt(a*b)
    s = s-2**(i+1)*(a**2-b**2)
    print(f"p_{i}={float(2*a**2/s):1.15f},
    ↪  |erreur_{i}|={float(abs(sympy.pi-2*a**2/s)):g}")
```

```
p_0=3.187672642712108, |erreur_0|=0.04608
p_1=3.141680293297653, |erreur_1|=8.76397e-05
p_2=3.141592653895446, |erreur_2|=3.05653e-10
p_3=3.141592653589793, |erreur_3|=3.71722e-21
p_4=3.141592653589793, |erreur_4|=5.4979e-43
p_5=3.141592653589793, |erreur_5|=1.20269e-86
```

Exercice A.10 (Calcul d'une intégrale par récurrence)
On souhaite calculer numériquement l'intégrale

$$\mathrm{I}_n = \int_0^1 x^n e^{\alpha x} \mathrm{d}x.$$

En intégrant par parties avec $f(x) = x^n$ et $g'(x) = e^{\alpha x}$, on obtient la relation de récurrence

$$\int x^n e^{\alpha x} \mathrm{d}x = x^n \frac{1}{\alpha} e^{\alpha x} - \frac{n}{\alpha} \int x^{n-1} e^{\alpha x} \mathrm{d}x \quad \rightsquigarrow \quad \mathrm{I}_n = \int_0^1 x^n e^{\alpha x} \mathrm{d}x = \frac{1}{\alpha} e^{\alpha} - \frac{n}{\alpha} \mathrm{I}_{n-1}.$$

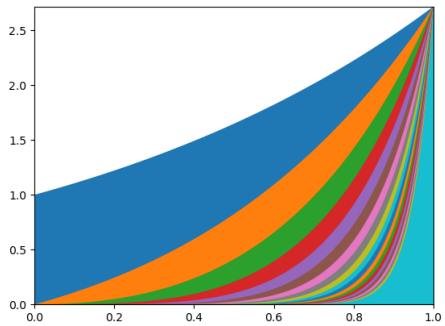
On en déduit la suite récurrente suivante

$$\begin{cases} \mathrm{I}_0 = \frac{e^{\alpha}-1}{\alpha}, \\ \mathrm{I}_{n+1} = \frac{1}{\alpha} e^{\alpha} - \frac{n+1}{\alpha} \mathrm{I}_n, \quad \text{pour } n \in \mathbb{N}. \end{cases}$$

Écrire un programme pour calculer cette suite avec $n = 50$ et comparer le résultat numérique avec la limite théorique $\mathrm{I}_n \xrightarrow[n \to +\infty]{} 0$.

Correction
Considérons $\alpha = 1$. On commence par visualiser l'évolution de $\{\mathrm{I}_n\}_n$ pour différentes valeurs de n. On observe que $\mathrm{I}_{n+1} < \mathrm{I}_n$.

```
import numpy as np
import matplotlib.pyplot as plt

alpha = 1
x = np.linspace(0,1,101)
plt.axis([0,1,0,max(1,np.exp(alpha))])
line, = plt.plot([],[],lw=2)

for n in range(20):
    plt.fill_between(x,0,x**n*np.exp(alpha*x))
plt.show()
```



Si on calcule I_n à l'aide de la formule de récurrence, on remarque que $0 < \mathrm{I}_{n+1} < \mathrm{I}_n$ pour $n < 17$, mais qu'à $n = 17$ on a $\mathrm{I}_{17} < 0$ et la suite devient instable et la suite obtenue numériquement ne tend pas vers zéro comme attendu. Ce comportement est généralisable à d'autres valeurs de α.

```python
from math import exp
alpha = 1
I = [(exp(alpha)-1)/alpha]
print(f"I_{0}={I[-1]}")
for n in range(20):
    I.append(exp(alpha)/alpha -(n+1)*I[-1]/alpha)
    print(f"I_{n} = {I[-1]}")
```

```
I_0=1.718281828459045
I_0 = 1.0
I_1 = 0.7182818284590451
I_2 = 0.5634363430819098
I_3 = 0.4645364561314058
I_4 = 0.395599547802016
I_5 = 0.34468454164694906
I_6 = 0.30549003693040166
I_7 = 0.27436153301583177
I_8 = 0.24902803131655915
I_9 = 0.22800151529345358
I_10 = 0.21026516023105568
I_11 = 0.19509990568637692
I_12 = 0.18198305453614516
I_13 = 0.17051906495301283
I_14 = 0.1604958541638526
I_15 = 0.15034816183740363
I_16 = 0.16236307722318344
I_17 = -0.2042535615582568
I_18 = 6.599099498065924
I_19 = -129.26370813285942
```

L'instabilité provient de la propagation des erreurs d'arrondi. En passant de I_n à I_{n+1}, l'erreur numérique est multipliée par n :

$$\begin{aligned}
\varepsilon_{n+1} &= \mathrm{I}_{n+1}^{\text{exacte}} - \mathrm{I}_{n+1}^{\text{approx}} \\
&= \left(\frac{1}{\alpha}e^{\alpha} - \frac{n+1}{\alpha}\mathrm{I}_n^{\text{exacte}}\right) - \left(\frac{1}{\alpha}e^{\alpha} - \frac{n+1}{\alpha}\mathrm{I}_n^{\text{approx}}\right) \\
&= -\frac{n+1}{\alpha}(\mathrm{I}_n^{\text{exacte}} - \mathrm{I}_n^{\text{approx}}) \\
&= -\frac{n+1}{\alpha}\varepsilon_n
\end{aligned}$$

Ainsi, l'erreur croit comme $\frac{n!}{\alpha^n}|\varepsilon_0|$, expliquant l'instabilité.

NB : ici on ne peut pas utiliser le module `fraction` car $e \notin \mathbb{Q}$.

cf. https://scipython.com/book/chapter-9-general-scientific-programming/examples/numerical-stability-of-an-integral-solved-by-recursion/

Pour éviter cela, il est préférable d'utiliser l'expression directe de I_n.

Bonus ($\alpha = -1$) : la suite (I_n) définie par

$$\begin{cases} \mathrm{I}_0 = 1 - \frac{1}{e} \\ \mathrm{I}_{n+1} = (n+1)\mathrm{I}_n - \frac{1}{e} \end{cases}$$

peut être explicitée par la formule

$$\mathrm{I}_n = \frac{n!}{e}\left(e - \sum_{k=0}^{n}\frac{1}{k!}\right)$$

En effet, on a

$$\frac{\mathrm{I}_{n+1}}{(n+1)!} - \frac{\mathrm{I}_n}{n!} = -\frac{1}{e}\frac{1}{(n+1)!} \quad \rightsquigarrow \quad \frac{\mathrm{I}_n}{n!} = \mathrm{I}_0 + \sum_{h=0}^{n}\left(\frac{\mathrm{I}_{h+1}}{(h+1)!} - \frac{\mathrm{I}_h}{h!}\right) = \mathrm{I}_0 - \frac{1}{e}\sum_{k=0}^{n}\frac{1}{k!} \quad \rightsquigarrow \quad \frac{\mathrm{I}_n}{n!} = 1 - \frac{1}{e}\sum_{k=0}^{n}\frac{1}{k!}.$$

On a bien $\lim_{n\to\infty}\mathrm{I}_n = 0$ car

$$\begin{aligned}
n!\sum_{k=n+1}^{\infty}\frac{1}{k!} &= n!\left(\frac{1}{(n+1)!} + \frac{1}{(n+2)!} + \frac{1}{(n+3)!} + \dots\right) = \frac{1}{n+1}\left(1 + \frac{1}{n+2} + \frac{1}{(n+2)(n+3)} + \dots\right) \\
&\le \frac{1}{n+1}\left(1 + \frac{1}{n+2} + \left(\frac{1}{n+2}\right)^2 + \dots\right) = \frac{n+2}{(n+1)^2} \to 0.
\end{aligned}$$

Remarque : $\mathrm{I}_n = \int_0^1 x^n e^{-x}\,dx$ d'où par parties $\mathrm{I}_{n+1} = n\mathrm{I}_n - \mathrm{I}_n$.

Exercice A.11 (Évaluer la fonction de Rump)

Évaluer au point $(x, y) = (77617, 33096)$ la fonction de deux variables suivante :

$$f(x,y) = \frac{1335}{4}y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + \frac{11}{2}y^8 + \frac{x}{2y}$$

Correction

```
>>> f = lambda x,y : 1335*y**6/4+x**2*(11*x**2*y**2-y**6-121*y**4-2) + 11*y**8/2+x/(2*y)
>>> print(f(77617,33096))
1.1726039400531787
```

Si on fait le calcul à la main ou on utilise le module `fractions` ou encore le module `sympy` (pour le calcul formel) comme ci-dessous, on trouve une valeur exacte d'environ -0.8273960599 : non seulement l'ordinateur a calculé une valeur très éloignée du résultat mais en plus, le signe n'est même pas le bon.

```
>>> from fractions import Fraction
>>> f = lambda x,y : Fraction(1335,4)*y**6 + x**2*(11*x**2*y**2-y**6-121*y**4-2) \
...                 + Fraction(11,2)*y**8+Fraction(x,2*y)
>>> sol=f(77617,33096)
>>> print(sol,"=",float(sol))
-54767/66192 = -0.8273960599468214

>>> import sympy
>>> sympy.var('x,y')
(x, y)
>>> g = 1335*y**6/4+x**2*(11*x**2*y**2-y**6-121*y**4-2) + 11*y**8/2+x/(2*y)
>>> sol=g.subs({x:77617, y:33096})
>>> print(sol,"=",sol.evalf())
-54767/66192 = -0.827396059946821
```

Exercice Bonus A.12 (Approximations de fonctions polynomiales)

Considérons les fonctions polynomiales suivantes :

$$\begin{aligned} f(x) &= x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1, \\ h(x) &= ((((((x-7)x+21)x-35)x+35)x-21)x+7)x-1, \\ g(x) &= (x-1)^7. \end{aligned}$$

Ces trois fonctions sont mathématiquement égales. Cependant, lorsqu'elles sont tracées dans l'intervalle $0.988 \leq x \leq 1.012$, des différences significatives apparaissent. Expliquer l'origine de ces différences.

Correction

La fonction f est évaluée de manière approchée à proximité de $x = 1$. Les calculs successifs impliquent des opérations d'addition et de soustraction sur des termes dont les valeurs absolues peuvent atteindre des ordres de grandeur proches de 35×1024^4. Cela provoque une perte de précision en raison des erreurs d'arrondi, ce qui explique les anomalies observées dans le tracé de f.

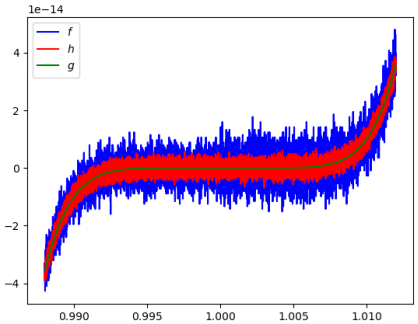
De manière similaire, la fonction h est calculée suivant une forme développée par la méthode de Horner, qui est souvent plus stable numériquement. Cependant, elle reste sujette aux mêmes limitations de précision dues à l'accumulation d'erreurs lors des multiplications successives.

En revanche, la fonction g, exprimée sous forme factorisée $(x-1)^7$, est calculée avec une meilleure précision, car elle minimise le nombre d'opérations et réduit l'effet des erreurs d'arrondi. C'est pourquoi $g(x)$ offre une représentation plus fidèle dans cet intervalle.

```
import numpy as np

f = lambda x: x**7-7*x**6+21*x**5-35*x**4+35*x**3-21*x**2+7*x-1
h = lambda x: ((((((x-7)*x+21)*x-35)*x+35)*x-21)*x+7)*x-1
g = lambda x: (x-1)**7

# Affichage des valeurs de f et g pour x ≈ 1
from tabulate import tabulate
T = []
T.append(['x', 'f(x)', 'g(x)', 'h(x)'])
for x in np.linspace(0.988, 1.012, 11):
    T.append( [x, f(x) , g(x), h(x)] )
print(tabulate(T, headers='firstrow', floatfmt='1.4e'))
```

```
# Affichage des graphes de f et g pour x ≈ 1
import matplotlib.pyplot as plt
x = np.linspace(0.988, 1.012, 10000)
plt.plot(x, f(x), label=r'$f$', color='blue')
plt.plot(x, h(x), label=r'$h$',color='red')
plt.plot(x, g(x), label=r'$g$',color='green')
# plt.ylim(0.99995, 1.00005)
plt.legend()
plt.savefig("Images/exo-poly.png")
plt.show()

# Vérifions l'égalité des deux fonctions
import sympy as sp
sp.var('x')
print("Avec sympy: f(x)-g(x) = ", sp.simplify(f(x)-g(x)))
```

```
          x          f(x)          g(x)          h(x)
  ----------  -----------  -----------  -----------
  9.8800e-01  -3.5527e-14  -3.5832e-14  -3.6859e-14
  9.9040e-01  -1.1546e-14  -7.5145e-15  -7.5495e-15
  9.9280e-01  -3.5527e-15  -1.0031e-15  -2.1094e-15
  9.9520e-01   3.5527e-15  -5.8707e-17  -2.6645e-15
  9.9760e-01   7.1054e-15  -4.5865e-19   4.4409e-16
  1.0000e+00   0.0000e+00   0.0000e+00   0.0000e+00
  1.0024e+00  -1.4211e-14   4.5865e-19   8.8818e-16
  1.0048e+00   0.0000e+00   5.8707e-17   8.8818e-16
  1.0072e+00   8.8818e-16   1.0031e-15   8.8818e-16
  1.0096e+00   4.4409e-15   7.5145e-15   7.5495e-15
  1.0120e+00   4.6185e-14   3.5832e-14   3.5527e-14
  Avec sympy: f(x)-g(x) =  0
```



Exercice Bonus A.13 (Approximations de deux fonctions trigonométriques)

Les fonctions

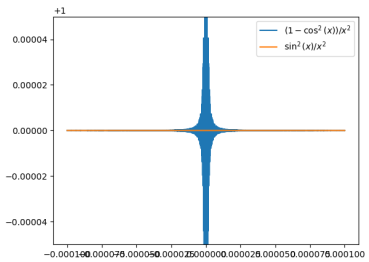
$$f(x) = \frac{1-\cos^2(x)}{x^2}, \qquad \text{et} \qquad g(x) = \frac{\sin^2(x)}{x^2}$$

sont égales mais tracées dans l'intervalle $-0.001 \leq x \leq 0.001$ montrent une différence significative. Expliquer l'origine de cette différence.

Source : https://scipython.com/book/chapter-9-general-scientific-programming/questions/floating-point-approximations-to-two-trigonometric-functions/

Correction

L'expression `1-np.cos(x)**2` est calculée de façon approchée à proximité de $x = 0$, ce qui entraîne une perte de précision et des oscillations sauvages dans le tracé de f.




```
import numpy as np

f = lambda x: (1 - np.cos(x)**2)/x**2
g = lambda x: (np.sin(x)/x)**2
```

```python
# Affichage des valeurs de f et g pour x ≈ 0
from tabulate import tabulate
T = []
T.append(['x', '1 - np.cos(x)**2' , 'np.sin(x)**2' ,  'f(x)', 'g(x)'])
for x in np.linspace(-1.e-7, 1.e-7, 10):
    T.append( [x, 1 - np.cos(x)**2 , np.sin(x)**2, f(x) , g(x)] )
print(tabulate(T, headers='firstrow', floatfmt='1.10e'))

# Affichage des graphes de f et g pour x ≈ 0
import matplotlib.pyplot as plt
x = np.linspace(-1.e-4, 1.e-4, 10000)
plt.plot(x, f(x), label=r'$(1-\cos^2(x))/x^2$')
plt.plot(x, g(x), label=r'$\sin^2(x)/x^2$')
plt.ylim(0.99995, 1.00005)
plt.legend()
# plt.savefig("exo-PB.png")
plt.show()
```

```
                x    1 - np.cos(x)**2      np.sin(x)**2              f(x)              g(x)
-----------------  ------------------  ----------------  ----------------  ----------------
-1.0000000000e-07    9.9920072216e-15  1.0000000000e-14  9.9920072216e-01  1.0000000000e+00
-7.7777777778e-08    5.9952043330e-15  6.0493827160e-15  9.9104398157e-01  1.0000000000e+00
-5.5555555556e-08    3.1086244690e-15  3.0864197531e-15  1.0071943279e+00  1.0000000000e+00
-3.3333333333e-08    1.1102230246e-15  1.1111111111e-15  9.9920072216e-01  1.0000000000e+00
-1.1111111111e-08    2.2204460493e-16  1.2345679012e-16  1.7985612999e+00  1.0000000000e+00
 1.1111111111e-08    2.2204460493e-16  1.2345679012e-16  1.7985612999e+00  1.0000000000e+00
 3.3333333333e-08    1.1102230246e-15  1.1111111111e-15  9.9920072216e-01  1.0000000000e+00
 5.5555555556e-08    3.1086244690e-15  3.0864197531e-15  1.0071943279e+00  1.0000000000e+00
 7.7777777778e-08    5.9952043330e-15  6.0493827160e-15  9.9104398157e-01  1.0000000000e+00
 1.0000000000e-07    9.9920072216e-15  1.0000000000e-14  9.9920072216e-01  1.0000000000e+00
```

Exercices

1.1. Exercice (Devine le résultat – affectations) . . . 21
1.2. Exercice (Devine le résultat – affectations) . . . 21
1.3. Exercice (Séquentialité) . . . 22
1.4. Exercice (Devine le résultat – logique) . . . 22
1.5. Exercice (Mode interactif) . . . 22
1.6. Exercice (CodEx: Autour des booléens) . . . 23
1.7. Exercice (Mode script) . . . 24
1.9. Exercice (Évaluation numérique d'une fraction) . . . 25
1.10. Exercice (Conversion j/h/m/s ⇝ s) . . . 25
1.11. Exercice (Conversion s ⇝ j/h/m/s) . . . 26
1.13. Exercice (Paradoxe de Simpson: la difficulté de comparer des ratios) . . . 26

2.1. Exercice (Devine le résultat – ASCII art) . . . 52
2.2. Exercice (Devine le résultat – affectations et print) . . . 52
2.3. Exercice (Cocorico) . . . 52
2.4. Exercice (Devine le résultat - String: + et *) . . . 52
2.5. Exercice (String concatenation) . . . 53
2.6. Exercice (Devine le résultat – opérations et conversions de types) . . . 53
2.7. Exercice (Opérations avec différents types) . . . 53
2.8. Exercice (Calculer l'age) . . . 54
2.9. Exercice (Happy Birthday) . . . 54
2.10. Exercice (Compter le nombre de caractères blancs) . . . 55
2.11. Exercice (Nombre de chiffres de l'écriture d'un entier) . . . 55
2.12. Exercice (Devine le résultat – `string`) . . . 56
2.13. Exercice (Sous-chaînes de caractères) . . . 56
2.14. Exercice (Chaîne de caractères palindrome) . . . 57
2.15. Exercice (Nombre palindrome) . . . 57
2.20. Exercice (Devine le résultat – Yoda) . . . 60
2.21. Exercice (Classement d'une course cycliste) . . . 60
2.22. Exercice (Listes et sous-listes) . . . 61
2.23. Exercice (Devine le résultat) . . . 62
2.24. Exercice (Saisons (liste de listes de string)) . . . 62
2.25. Exercice (Insertions) . . . 63
2.26. Exercice (Nombres pairs) . . . 64
2.27. Exercice (Moyenne) . . . 64
2.28. Exercice (Effectifs et fréquence) . . . 64
2.29. Exercice (Max-Min) . . . 65
2.30. Exercice (Range) . . . 65
2.31. Exercice (Note ECUE) . . . 66
2.32. Exercice (texte ⇝ liste de mots) . . . 66
2.33. Exercice (Devine le résultat - tuples) . . . 66
2.34. Exercice (Liste de tuples) . . . 67
2.35. Exercice (Manipulation de matrices avec des listes imbriquées) . . . 68
2.36. Exercice (Robots) . . . 68
2.37. Exercice (Pièges avec les copies et alias en Python) . . . 70
2.38. Exercice (Devine le résultat) . . . 73

3.1. Exercice (Importance de l'indentation) . . . 79
3.2. Exercice (Devine le résultat) . . . 79
3.3. Exercice (Blanche Neige) . . . 79
3.4. Exercice (Années bissextiles) . . . 80
3.5. Exercice (Détermination de l'état de l'eau en fonction de la température) . . . 82
3.6. Exercice (Calculer $|x|$) . . . 82
3.7. Exercice (Calcul de l'Indice de Masse Corporelle (IMC)) . . . 82
3.8. Exercice (Note ECUE) . . . 83

3.9. Exercice ($ax^2 + bx + c = 0$) . . . 83
3.10. Exercice (Triangles) . . . 84

4.1. Exercice (Devine le résultat - `for`) . . . 95
4.2. Exercice (Mes ingrédients préférés) . . . 95
4.3. Exercice (Devine le résultat) . . . 96
4.6. Exercice (Pièces magiques) . . . 97
4.7. Exercice (Lower to Upper) . . . 98
4.8. Exercice (Poids sur la Lune) . . . 98
4.9. Exercice (Vente de voitures) . . . 98
4.10. Exercice (Multiplication sans "*") . . . 99
4.11. Exercice (Triangles) . . . 99
4.12. Exercice (Nombre de voyelles) . . . 99
4.13. Exercice (Cartes de jeux) . . . 100
4.14. Exercice (Tables de multiplication) . . . 100
4.16. Exercice (Parcourir deux listes) . . . 102
4.17. Exercice (Parcourir deux chaînes de caractères) . . . 102
4.18. Exercice (Devine le résultat - `while`) . . . 103
4.19. Exercice (CodEx: Premier minimum local) . . . 103
4.20. Exercice (Suites type ①: $u_n = f(n)$. Recherche de seuil: plus petit n tel que...) . . . 104
4.21. Exercice (Suites type ①: $u_n = f(n)$ (suite géométrique)) . . . 105
4.22. Exercice (Recherche de seuil: plus petit n tel que...) . . . 106
4.23. Exercice (Recherche de seuil: méthode du rectangle à gauche) . . . 106
4.24. Exercice (Suite type ②: récurrence $u_{n+1} = f(u_n)$) . . . 107
4.25. Exercice (Suites type ②: $\sqrt{2\sqrt{2\sqrt{2\sqrt{2\ldots}}}}$) . . . 108
4.26. Exercice (Suites type ②: $\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+\ldots}}}}$) . . . 108
4.27. Exercice (Suites type ③: récurrence à deux pas $u_{n+1} = f(u_n, u_{n-1})$ (Fibonacci)) . . . 109
4.28. Exercice (Suites type ③: récurrence à deux pas $u_{n+1} = f(u_n, u_{n-1})$ (Fibonacci – bis)) . . . 110
4.30. Exercice (Défi Turing n°2 – Fibonacci) . . . 111
4.31. Exercice (Suites type ③: récurrence à deux pas $u_{n+1} = f(u_n, u_{n-1})$ (Euclide)) . . . 111
4.32. Exercice (Suites type ④: récurrence $(u, v)_{n+1} = (f_1(u_n, v_n), f_2(u_n, v_n))$) . . . 112
4.33. Exercice (Suites type ⑤: récurrence $(u, v)_{n+1} = (f_1(n, u_n, v_n), f_2(n, u_n, v_n))$) . . . 113
4.34. Exercice (La suite de Stern-Brocot) . . . 113
4.35. Exercice (La suite G de Hofstadter) . . . 114
4.36. Exercice (La suite H de Hofstadter) . . . 114
4.37. Exercice (La suite Female-Male de Hofstadter) . . . 115
4.38. Exercice (La suite Hofstadter Figure-Figure (R et S)) . . . 115
4.41. Exercice (S-T-R-E-T-C-H I-T O-U-T) . . . 116
4.42. Exercice (Nombre mystère) . . . 117
4.43. Exercice (4n) . . . 118
4.66. Exercice (Cadeaux) . . . 129
4.67. Exercice (Affichage) . . . 129
4.69. Exercice (Maximum d'une liste de nombres sans la fonction `max`) . . . 130
4.70. Exercice (Chaîne la plus longue d'une liste de strings) . . . 130
4.74. Exercice (Entier palindrome dans une base b) . . . 132

5.1. Exercice (Sous-listes) . . . 141
5.2. Exercice (Somme des carrés) . . . 141
5.3. Exercice (Œufs) . . . 142
5.4. Exercice (Rallye mathématique de la Réunion 2005, exercice 2) . . . 142
5.6. Exercice (Pièces magiques - bis) . . . 143
5.7. Exercice (Temperature moyenne) . . . 143
5.8. Exercice (Liste de `str` ⇝ liste de `int`) . . . 144
5.9. Exercice (Chaîne de caractères ⇝ liste de `str`) . . . 144
5.10. Exercice (Somme des chiffres d'un nombre) . . . 144
5.11. Exercice (Défi Turing n°5 – somme des chiffres d'un nombre) . . . 145
5.12. Exercice (Liste de Moyennes) . . . 145

5.13. Exercice (Diviser une liste en sous-listes) . . . 145
5.14. Exercice (`argmin`: position du minimum d'une liste de nombres) . . . 145
5.15. Exercice (Chaturanga) . . . 146
5.16. Exercice (Filtrer une liste) . . . 147
5.17. Exercice (Liste des diviseurs) . . . 147
5.21. Exercice (Soustraire deux listes) . . . 149
5.22. Exercice (Jours du mois) . . . 149
5.23. Exercice (Conversion de températures) . . . 149
5.24. Exercice (Années bissextiles) . . . 150
5.26. Exercice (Vacances) . . . 151
5.27. Exercice (Produit matriciel) . . . 151
5.28. Exercice (Nombres triangulaires) . . . 152
5.31. Exercice (Table de multiplication) . . . 154
5.32. Exercice (Défi Turing n°1 – somme de multiples) . . . 155
5.33. Exercice (Nombres de Armstrong) . . . 156
5.34. Exercice (Imprimeur et nombre d'utilisations d'un chiffres) . . . 156
5.35. Exercice (Défi Turing n°85 – Nombres composés de chiffres différents) . . . 157
5.38. Exercice (`sum() len() all() any()`) . . . 157
5.39. Exercice (Sélection de nombres) . . . 158

6.1. Exercice (Devine le résultat - bêtisier) . . . 175
6.2. Exercice (Devine le résultat - variables globales *vs* locales) . . . 175
6.4. Exercice (Devine le résultat) . . . 178
6.5. Exercice (Renvoyer un booléen) . . . 178
6.6. Exercice (F: $\mathbb{R} \to \mathbb{R}^2$) . . . 179
6.8. Exercice (Calcul de l'effectif et de la fréquence – démarche guidée) . . . 180
6.9. Exercice (CodEx: Rendement d'un livret – démarche guidée) . . . 182
6.10. Exercice (Mes premières potions magiques avec `def`) . . . 183
6.11. Exercice (Mes premières fonctions λ) . . . 185
6.12. Exercice (Jours de la semaine) . . . 187
6.13. Exercice (Pièces magiques - ter) . . . 187
6.14. Exercice (Le problème de la ferme) . . . 187
6.15. Exercice (Poids sur la Lune) . . . 188
6.16. Exercice (Prix d'un billet) . . . 188
6.17. Exercice (Swap) . . . 189
6.18. Exercice (CodEx - Casseur de digicode: fonction de fonction) . . . 189
6.20. Exercice (Cul de Chouette - version simplifiée) . . . 191
6.21. Exercice (Nombres de Friedman) . . . 191
6.22. Exercice (CodEx - Vérification de multiples) . . . 192
6.23. Exercice (CodEx - Occurrences d'un caractère dans un mot) . . . 192
6.24. Exercice (CodEx - Occurrences d'un élément dans un tableau) . . . 193
6.25. Exercice (CodEx - Conversion en minuscules) . . . 193
6.26. Exercice (CodEx - La moyenne olympique) . . . 194
6.27. Exercice (CodEx: Différences entre deux tableaux) . . . 194
6.28. Exercice (Vérifier si une matrice est antisymétrique) . . . 195
6.29. Exercice ($u_{n+1} = f(n, u_n)$) . . . 196
6.30. Exercice (CodEx: détermination d'un seuil, suite de type ①) . . . 196
6.31. Exercice (Recherche de seuil: plus petit diviseur) . . . 197
6.32. Exercice (CodEx: détermination d'un seuil, suite de type ②) . . . 197
6.33. Exercice (Détermination d'un seuil, suite de type ②) . . . 197
6.34. Exercice (Suites type ③: récurrence à deux pas $u_{n+1} = f(u_n, u_{n-1})$) . . . 199
6.35. Exercice (Moyenne Arithmétique-Géométrique et Constante de Gauss) . . . 199
6.36. Exercice (Schémas d'analyse numérique: recherche de zéros) . . . 200
6.37. Exercice (Aucune Condition) . . . 201
6.38. Exercice (Recréer la Fonction `abs()`) . . . 201
6.39. Exercice (Vérification de Doublons) . . . 202
6.40. Exercice (Entier manquant) . . . 202
6.41. Exercice (Pair ou Impair) . . . 202
6.42. Exercice (Divisibilité) . . . 203

6.43. Exercice (Liste impairs) . . . 203
6.44. Exercice (Liste divisibles) . . . 204
6.45. Exercice (Nombres parfaits) . . . 204
6.46. Exercice (Radars routiers) . . . 205
6.47. Exercice (Multiple de 10 le plus proche) . . . 205
6.48. Exercice (Rugby) . . . 206
6.50. Exercice (Mélange à queue d'aronde) . . . 208
6.51. Exercice (CodEx: Redimensionner un tableau) . . . 208
6.54. Exercice (Masquer tickets de caisse) . . . 210
6.56. Exercice (CodEx - Nombre de zéros à la fin d'un entier sans `str`) . . . 211
6.57. Exercice (Liste qui transforme en 0 les termes à partir du premier 0) . . . 211
6.58. Exercice (Password) . . . 212
6.59. Exercice (Distance de Hamming) . . . 212
6.60. Exercice (Nombre de Harshad) . . . 213
6.61. Exercice (Validité d'un code de photocopieuse) . . . 213
6.62. Exercice (Numéro ISBN-10) . . . 214
6.63. Exercice (Numéro de sécurité sociale) . . . 214
6.65. Exercice (Algorithme d'Euclide) . . . 215
6.67. Exercice (Montagnes/vallées: monotonie dans une séquence) . . . 217
6.68. Exercice (Détection de sommets dans un profil de montagne) . . . 218
6.69. Exercice (CodEx: Génération de cartes d'anniversaire pour chats) . . . 219
6.70. Exercice (Rendu monnaie) . . . 220
6.72. Exercice (Can Balance) . . . 221
6.74. Exercice (Nombre miroir) . . . 222
6.75. Exercice (Point milieu) . . . 223
6.76. Exercice (Normaliser une liste) . . . 223
6.77. Exercice (Couper un intervalle) . . . 224
6.78. Exercice (Représentation et manipulation de polynômes) . . . 225
6.79. Exercice (Triangle de Pascal) . . . 228
6.80. Exercice (Voyelles) . . . 228
6.81. Exercice (CodEx: Élocution chez le dentiste) . . . 229
6.82. Exercice (Pangramme) . . . 229
6.83. Exercice (Stock Profit) . . . 230
6.84. Exercice (OVNI) . . . 231
6.85. Exercice (CodEx: Bâtiments éclairés au soleil couchant) . . . 231
6.86. Exercice (CodEx: Bâtiments éclairés au soleil levant) . . . 232
6.87. Exercice (Plus grande surface rectangulaire dans un histogramme) . . . 232
6.88. Exercice (Calcul de l'aire d'union de rectangles superposés) . . . 233
6.89. Exercice (Nombres premiers) . . . 235
6.95. Exercice (Comptage des doigts) . . . 237
6.100. Exercice (Code César) . . . 240
6.107. Exercice (Palindromes) . . . 246
6.112. Exercice (Conversion base 2 → base 10) . . . 248
6.113. Exercice (Conversion base 3 → base 10) . . . 249
6.114. Exercice (Écriture d'un entier dans une base quelconque et entiers brésiliens) . . . 249
6.115. Exercice (Spreadsheet Column Number) . . . 250

7.1. Exercice (Is the Water Boiling?) . . . 277
7.2. Exercice (Benford's Law for the Second Digit) . . . 277
7.3. Exercice (Module `math` – π, e, $\sqrt{}$) . . . 278
7.4. Exercice (Module `math` – sin, cos, tan, π, e) . . . 278
7.6. Exercice (Module `math` - Angles remarquables) . . . 279
7.7. Exercice (Module `math` - Paper folding) . . . 280
7.8. Exercice (Module `math` – Approximation de la valeur ponctuelle de dérivées) . . . 281
7.9. Exercice (Module `math` – Factorielle) . . . 281
7.10. Exercice (Module `math` – Factorion) . . . 282
7.11. Exercice (Module `math` – Stirling) . . . 282
7.12. Exercice (Détermination d'un seuil, suite de type ①) . . . 283
7.13. Exercice (Module `math` – Projet Euler n°20 et Défi Turing n°6 – somme de chiffres) . . . 283

7.14. Exercice (Module `math` – Nombres de Brown) . . . 284
7.15. Exercice (Module `math` – Approximation de e) . . . 284
7.16. Exercice (Modules `math` et `Decimal` – Polignac) . . . 285
7.17. Exercice (Suites type ③ : comparaison itératif / formule explicite) . . . 286
7.18. Exercice (Module `math` – Indicatrice d'Euler) . . . 288
7.19. Exercice (Module `math` – Énoncer des chiffres) . . . 288
7.20. Exercice (Module `math` – Overflow) . . . 288
7.21. Exercice (Module `math` – Distance) . . . 289
7.22. Exercice (Module `math` – Distance point droite) . . . 291
7.27. Exercice (Temps de parcours d'une liste) . . . 296
7.28. Exercice (Réduire une fraction : module `math` puis modules `fractions` et `sympy`) . . . 297
7.30. Exercice (Module `random` - Jeu de dé) . . . 299
7.31. Exercice (Module `random` - Tirage de nombres aléatoires) . . . 299
7.32. Exercice (Module `random` - Calcul de fréquences) . . . 300
7.33. Exercice (Module `random` - Puce) . . . 300
7.36. Exercice (Module `math` – Cylindre) . . . 302
7.37. Exercice (Module `math` – Formule d'Héron) . . . 302
7.38. Exercice (Module `math` – Formule de Kahan) . . . 303
7.39. Exercice (Module `math` – Cercle circonscrit) . . . 303
7.41. Exercice (Module `random` - Distance moyenne entre deux points aléatoires d'un carré) . . . 305
7.42. Exercice (Module `random` - Kangourou) . . . 305
7.43. Exercice (Module `math` – sin cos) . . . 306
7.44. Exercice (Approximations de ln) . . . 306
7.45. Exercice (Approximations de π) . . . 307
7.46. Exercice (Module `scipy` - Calcul approché d'une intégrale (méthode Monte-Carlo)) . . . 314
7.47. Exercice (Module `numpy` - Statistique) . . . 315
7.48. Exercice (Module `numpy` - Statistique et suites) . . . 316

8.1. Exercice (Le gateau) . . . 335
8.2. Exercice (Bhaskara I) . . . 335
8.3. Exercice (Tracer des droites) . . . 336
8.4. Exercice (Tracer plusieurs courbes) . . . 337
8.5. Exercice (Encadrement de e) . . . 337
8.6. Exercice (Tracer une fonction définie par morceaux) . . . 338
8.7. Exercice (Représentation graphique des racines carrées) . . . 339
8.8. Exercice (Construction alternée des racines carrées) . . . 339
8.9. Exercice (Diagramme de Farey) . . . 342
8.10. Exercice (CodEx : Triangle mystérieux) . . . 342
8.11. Exercice (Module `scipy` - Calcul approché d'une intégrale) . . . 343
8.12. Exercice (Dépréciation ordinateur) . . . 344
8.13. Exercice (Mercato) . . . 345
8.14. Exercice (Résolution graphique d'une équation) . . . 345
8.15. Exercice (Résolution graphique) . . . 346
8.16. Exercice (Courbe paramétrée) . . . 347
8.17. Exercice (Courbe paramétrée) . . . 348
8.18. Exercice (Courbe polaire) . . . 348
8.19. Exercice (Courbe polaire) . . . 348
8.20. Exercice (Proies et prédateurs) . . . 349
8.21. Exercice (Coïncidences et anniversaires) . . . 351
8.22. Exercice (Conjecture de Syracuse) . . . 353
8.25. Exercice (Notation cistercienne) . . . 358

A.1. Exercice (Devine le résultat) . . . 375
A.2. Exercice (Qui est plus grand ?) . . . 375
A.3. Exercice (Équation quadratique) . . . 376
A.4. Exercice (Integer VS Floating point number) . . . 377
A.5. Exercice (Un contre-exemple du dernier théorème de Fermat ?) . . . 377
A.6. Exercice (Suites) . . . 378
A.7. Exercice (Suite de Muller) . . . 380

A.8. Exercice (Défi Turing n°86 – Le curieux 2 000-ème terme) 381
A.9. Exercice (Algorithme de Brent et Salamin) 381
A.10. Exercice (Calcul d'une intégrale par récurrence) 382
A.11. Exercice (Évaluer la fonction de Rump) 383

Exercices ★

1.8. Exercice Bonus (Rendre un script exécutable : la ligne *shebang*) . . . 24
1.12. Exercice Bonus (Années martiennes — *cf.* cours N. MELONI) . . . 26

2.16. Exercice Bonus (Tableau d'amortissement d'un prêt) . . . 58
2.17. Exercice Bonus (Format table) . . . 58
2.18. Exercice Bonus (Changement de casse & Co.) . . . 59
2.19. Exercice Bonus (Comptage, recherche et remplacement) . . . 59
2.39. Exercice Bonus (`str`⇝`list` : `split` et `join`) . . . 73
2.40. Exercice Bonus (Liste de tuples et Tuple de listes) . . . 73

3.11. Exercice Bonus (Pierre Feuille Ciseaux) . . . 85
3.12. Exercice Bonus (Pierre, feuille, ciseaux, lézard, Spock) . . . 86

4.4. Exercice Bonus (Devine le résultat – modification de listes) . . . 96
4.15. Exercice Bonus (Tables de vérité) . . . 101
4.44. Exercice Bonus (Défi Turing n°70 – Permutation circulaire) . . . 118
4.45. Exercice Bonus (Gator And Ducks) . . . 119
4.46. Exercice Bonus (Défis Turing n°72) . . . 120
4.47. Exercice Bonus (Dictionnaire) . . . 120
4.48. Exercice Bonus (Dictionnaires : construction d'un histogramme) . . . 120
4.49. Exercice Bonus (Dictionnaires : The Most Frequent) . . . 121
4.71. Exercice Bonus (Défi Turing n°9 – triplets pythagoriciens) . . . 130
4.72. Exercice Bonus (Défi Turing n°11 - nombre miroir) . . . 131
4.73. Exercice Bonus (Défis Turing n°13, n°43 et n°73 – Carrés parfaits palindromes) . . . 131
4.75. Exercice Bonus (Défi Turing 22 – Les anagrammes octuples) . . . 132
4.76. Exercice Bonus (Défi Turing n°21 – Bonne année 2013 !) . . . 132

5.18. Exercice Bonus (Défi Turing n°18 – Somme de nombres non abondants) . . . 148
5.19. Exercice Bonus (Défis Turing n°71 – ensembles) . . . 148
5.20. Exercice Bonus (Somme et de différence de deux carrés) . . . 148
5.25. Exercice Bonus (Tri personnalisé) . . . 150
5.29. Exercice Bonus (Nombres pentagonaux) . . . 153
5.30. Exercice Bonus (Défi Turing n°45 – Nombre triangulaire, pentagonal et hexagonal) . . . 154
5.42. Exercice Bonus (Défi Turing n° 40 – La constante de Champernowne) . . . 159
5.45. Exercice Bonus (Défi Turing n°29 – puissances distincts) . . . 161
5.46. Exercice Bonus (Défi Turing n°94 – Problème d'Euler n° 92) . . . 161
5.50. Exercice Bonus (Anniversaires) . . . 162
5.51. Exercice Bonus (Triplets pytagoriciens) . . . 163
5.52. Exercice Bonus (Dictionnaire ordonné) . . . 163

6.3. Exercice Bonus (Devine le résultat - Function scope) . . . 177
6.7. Exercice Bonus (Arguments nommés et arguments par défaut) . . . 179
6.19. Exercice Bonus (`map`) . . . 190
6.49. Exercice Bonus (Homer et la Duff) . . . 207
6.52. Exercice Bonus (Soirée parents-enfants) . . . 209
6.53. Exercice Bonus (Leet speak) . . . 209
6.64. Exercice Bonus (CheckSum) . . . 215
6.66. Exercice Bonus (Fusion de listes) . . . 216
6.73. Exercice Bonus (Matching Brackets) . . . 221
6.90. Exercice Bonus (Jumeaux) . . . 235
6.91. Exercice Bonus (Défi Turing n°96 – Projet d'Euler n°87) . . . 236
6.94. Exercice Bonus (Mot parfait) . . . 237
6.96. Exercice Bonus (Défi Turing n° 19 – Rencontre du quatrième type) . . . 238
6.98. Exercice Bonus (Défi Turing n° 52 – multiples constitués des mêmes chiffres) . . . 240
6.99. Exercice Bonus (Défi Turing n° 61 – Non à l'isolement !) . . . 240

6.101. Exercice Bonus (Défi Turing n° 17 – Nombres amicaux) . . . 241
6.102. Exercice Bonus (The Longest Word) . . . 242
6.103. Exercice Bonus (Tickets t+ RATP) . . . 242
6.104. Exercice Bonus (Tickets réseau Mistral) . . . 244
6.105. Exercice Bonus (Problème d'Euler n°9) . . . 245
6.108. Exercice Bonus (Défi Turing n°4 – nombre palindrome) . . . 247
6.116. Exercice Bonus (Suites de Kaprekar)) . . . 251
6.119. Exercice Bonus (Défi Turing n°141 – Combien de 6?) . . . 253
6.120. Exercice Bonus (Défi Turing n°33 – Pâques en avril) . . . 254
6.121. Exercice Bonus (Sun angle) . . . 255
6.122. Exercice Bonus (Angle entre les aiguilles d'une horloge) . . . 255
6.123. Exercice Bonus (Most Wanted Letter) . . . 256
6.124. Exercice Bonus (Bigger Price) . . . 257
6.125. Exercice Bonus (Sum by Types) . . . 257
6.126. Exercice Bonus (Common Words) . . . 258
6.127. Exercice Bonus (Flatten list) . . . 258
6.128. Exercice Bonus (Plan cyclique) . . . 259
6.129. Exercice Bonus (Systèmes Électoraux) . . . 261
6.130. Exercice Bonus (CodEx: Circulation de rumeurs) . . . 262

7.5. Exercice Bonus ($\sqrt{\tan(\pi)}$) . . . 279
7.23. Exercice Bonus (Module `math` – Distance point segment) . . . 293
7.24. Exercice Bonus (Module `math` – Distance point rectangle) . . . 294
7.25. Exercice Bonus (Module `math` – Longueur d'une portion de courbe) . . . 295
7.29. Exercice Bonus (Module `Fraction` – Triangles semblables) . . . 298
7.34. Exercice Bonus (Module `random` - Le nombre mystère) . . . 301
7.35. Exercice Bonus (Module `random` - Yahtzee) . . . 301
7.40. Exercice Bonus (Centre et rayon d'un cercle pour un arc donné) . . . 304
7.49. Exercice Bonus (Module `numpy` - Représentation et manipulation de polynômes) . . . 317
7.51. Exercice Bonus (Module `sympy` - Représentation et manipulation de polynômes) . . . 318
7.52. Exercice Bonus (Permutations avec itertools) . . . 319
7.53. Exercice Bonus (Permutations avec itertools) . . . 320
7.55. Exercice Bonus (Module `sympy` – devine le résultat) . . . 321
7.56. Exercice Bonus (Module `sympy` – calcul formel d'une dérivée) . . . 322
7.57. Exercice Bonus (Module `sympy` – composition de fonctions) . . . 322
7.58. Exercice Bonus (Module `sympy` – calcul des paramètres) . . . 322
7.59. Exercice Bonus (Module `sympy` – calcul des paramètres) . . . 322
7.60. Exercice Bonus (Module `sympy` – calcul de paramètres) . . . 323

8.23. Exercice Bonus (Cuve de fioul enterrée) . . . 355
8.24. Exercice Bonus (Le crible de Matiyasevitch) . . . 357
8.26. Exercice Bonus (Taux Marginal, Taux Effectif: comprendre les impôts) . . . 359
8.27. Exercice Bonus (Approximations d'intégrales) . . . 363

A.12. Exercice Bonus (Approximations de fonctions polynomiales) . . . 384
A.13. Exercice Bonus (Approximations de deux fonctions trigonométriques) . . . 385

Pydéfis

4.5. Exercice Bonus (Pydéfis – L'algorithme du professeur Guique) . . . 97
4.29. Exercice Bonus (Pydéfis – Fibonacci) . . . 110
4.39. Exercice Bonus (Pydéfis – La suite Q de Hofstadter) . . . 116
4.40. Exercice Bonus (La suite de Hofstadter-Conway) . . . 116
4.50. Exercice Bonus (Pydéfis – Vous parlez Fourchelangue ?) . . . 121
4.51. Exercice Bonus (Pydéfis - Code Konami) . . . 122
4.52. Exercice Bonus (Pydéfis - Difficile de comprendre un lapin crétin) . . . 122
4.53. Exercice Bonus (Pydéfis – Le retourneur de temps) . . . 122
4.54. Exercice Bonus (Pydéfis – Entrée au Ministère) . . . 123
4.55. Exercice Bonus (Pydéfis – Créatures nocturnes pas si sympathiques que cela...) . . . 123
4.56. Exercice Bonus (Pydéfis – SW I : À l'assaut de Gunray. Découpage de la porte blindée) . . . 124
4.57. Exercice Bonus (Pydéfis – L'hydre de Lerne) . . . 124
4.58. Exercice Bonus (Pydéfis – Le sanglier d'Érymanthe) . . . 125
4.59. Exercice Bonus (Pydéfis – Les dragées surprises) . . . 125
4.60. Exercice Bonus (Pydéfis – Les tubes à essai d'Octopus. Méli-mélo de poison) . . . 126
4.61. Exercice Bonus (Pydéfis – Le cours de potions) . . . 127
4.62. Exercice Bonus (Pydéfis – Désamorçage d'un explosif (I)) . . . 127
4.63. Exercice Bonus (Pydéfis – Méli Mélo de nombres) . . . 127
4.64. Exercice Bonus (Pydéfis – Suite Tordue) . . . 128
4.65. Exercice Bonus (Pydéfis – Bombe à désamorcer) . . . 128
4.68. Exercice Bonus (Pydéfis – Insaisissable matrice) . . . 129
4.77. Exercice Bonus (Pydéfis – L'escargot courageux) . . . 133
4.78. Exercice Bonus (Pydéfis – Mon beau miroir...) . . . 133
4.79. Exercice Bonus (Pydéfis – Persistance) . . . 133
4.80. Exercice Bonus (Pydéfis – Toc Boum) . . . 134
4.81. Exercice Bonus (Pydéfis – Les juments de Diomède) . . . 134
4.82. Exercice Bonus (Pydéfis – Produit et somme palindromiques) . . . 134

5.5. Exercice Bonus (Pydéfi – Le jardin des Hespérides) . . . 142
5.36. Exercice Bonus (Pydéfi – Piège numérique à Pokémons) . . . 157
5.37. Exercice Bonus (Pydéfi – Nos deux chiffres préférés) . . . 157
5.40. Exercice Bonus (Pydéfi – SW III : L'ordre 66 ne vaut pas 66...) . . . 159
5.41. Exercice Bonus (Pydéfi – Constante de Champernowne) . . . 159
5.43. Exercice Bonus (Pydéfi – Série décimée...) . . . 160
5.44. Exercice Bonus (Pydéfi – Désamorçage de bombe à distance (II)) . . . 160
5.47. Exercice Bonus (Pydéfi – Le pistolet de Nick Fury) . . . 161
5.48. Exercice Bonus (Pydéfi – Les nombres heureux) . . . 162
5.49. Exercice Bonus (Pydéfi – Le problème des boîtes à sucres) . . . 162

6.55. Exercice Bonus (PyDéfis – Les hybrides. Détection de bases inconnues) . . . 210
6.71. Exercice Bonus (PyDéfis – Monnaie) . . . 220
6.92. Exercice Bonus (PyDéfis – Premier particulier (1)) . . . 236
6.93. Exercice Bonus (Pydéfi – Einstein) . . . 237
6.97. Exercice Bonus (Pydéfi – Vif d'or) . . . 239
6.106. Exercice Bonus (PyDéfis – Cerbère) . . . 246
6.109. Exercice Bonus (PyDéfis – Le jour de la serviette) . . . 247
6.110. Exercice Bonus (Pydéfi – Les bœufs de Géryon) . . . 247
6.111. Exercice Bonus (Pydéfi – Le lion de Némée) . . . 248
6.117. Exercice Bonus (Pydéfi – Numération Gibi : le Neutubykw) . . . 253
6.118. Exercice Bonus (Pydéfi – Pokédex en vrac) . . . 253

7.26. Exercice Bonus (Pydéfis – La biche de Cyrénée) . . . 296
7.50. Exercice Bonus (PyDéfis – Les victimes de Tooms 1/2) . . . 318
7.54. Exercice Bonus (Pydéfis – Analyse de séquences 1/2) . . . 320