

Aide-mémoire Python

Initiation à la programmation avec Python – G. Faccanoni

Variables, types, opérateurs

- affectation : `x = 3` type : `type(x)`
- types de base : `int`, `float`, `str`, `bool` (`True/False`)
- `print(a, b, sep=..., end=...)` : affiche, séparateur et fin personnalisables
- arithmétique, `round/abs/divmod`, appartenance :

```
>>> print(1, 2, 3, sep="-")
1-2-3
>>> (2 + 3) * 4 / 2 - 1
9.0
>>> 7 // 2
3
>>> 7 % 2
1
>>> 2**10
1024

>>> round(3.14159, 2)
3.14
>>> abs(-5)
5
>>> divmod(7, 2)
(3, 1)
>>> 3 in [1,2,3]
True
```

- // division entière % reste ** puissance

comparaison	==	!=	<	>	<=	>=	
logique	and	or	not				
augmentés	+=	-=	*=	/=	//=	%=	**=

- ex. `x += 1` idem que `x = x + 1`

Chaînes de caractères (f-string)

```
>>> x = 20
>>> f"J'ai {x} ans"
"J'ai 20 ans"
>>> "J'ai " + str(x) + " ans"
"J'ai 20 ans"
>>> "ananas".count("a")
3
```

- `f"{x:.2f}"` (2 décimales)
- `f"{x = }"`
- + exige des chaînes des deux côtés : `str(x)` est obligatoire
- slicing : `s[i]`, `s[a:b:c]` (aussi pour listes, tuples)

```
s[0] s[1] s[2] s[3] s[4] s[5]
s = | P | Y | T | H | O | N |
s[-6] s[-5] s[-4] s[-3] s[-2] s[-1]
```

```
>>> s = "PYTHON"
>>> s[1:4]
'YTH'
>>> s[:2]
'PY'
>>> s[-3:]
'HON'
>>> s[::-1]
'NOHTYP'
>>> s[0:6:2]
'PTO'
```

Listes

```
>>> L = [1,2,3]
>>> len(L)
3
>>> L.append(4)
>>> L
[1, 2, 3, 4]
>>> L.extend([5,6])
>>> L
[1, 2, 3, 4, 5, 6]
>>> sorted([3,1,2])
[1, 2, 3]
>>> sum(L), max(L), min(L)
(21, 6, 1)
```

Une liste peut mélanger les types, y compris une sous-liste :

```
>>> M = [1, "deux", 3.0, True,
↪ [4,5]]
>>> M
[1, 'deux', 3.0, True, [4, 5]]
>>> M[4]
[4, 5]
>>> M[4][0]
4
```

Modifient `L` sur place (ne renvoient rien d'utile) :

- `L.append(x)`, `L.extend(L2)` : ajoutent à la fin
- `L.insert(i,x)` : insère `x` à l'indice `i`
- `L.remove(x)` : retire la 1^{ère} occurrence de `x`
- `L.pop(i)` : retire et renvoie l'élément d'indice `i`
- `L.sort()` : trie `L` `L.reverse()` : inverse `L`

Renvoient un résultat sans modifier `L` :

- `sorted(L)` : nouvelle liste triée
- `L.index(x)` : indice de la 1^{ère} occurrence de `x`
- `L[:]` : copie (superficielle) de `L` – attention, `L2 = L` ne copie pas !

Tuples

- création : `t = (1,2,3)` – non modifiable (immuable)
- empaquetage/dépaquetage : `t = 1,2` puis `a,b = t`
- échange de deux variables : `a, b = b, a`

Conversion de types (typecasting)

```
>>> int("3")
3
>>> float("3.5")
3.5
```

```
>>> str(3)
'3'
>>> bool(0)
False
>>> list((1,2,3))
[1, 2, 3]
>>> tuple([1,2,3])
(1, 2, 3)
```

Itérateur range

```
>>> list(range(3))
[0, 1, 2]
>>> list(range(1,4))
[1, 2, 3]
>>> list(range(0,6,2))
[0, 2, 4]
>>> list(enumerate(["a","b"]))
[(0, 'a'), (1, 'b')]
>>> list(zip([1,2],[3,4]))
[(1, 3), (2, 4)]
```

- `range(stop)`, `range(start,stop)`, `range(start,stop,step)`
- `enumerate(L)` : couples (indice, valeur)
- `zip(L1,L2)` : parcourt `L1` et `L2` en parallèle

Conditions

```
if cond:
...
elif cond2:
...
else:
...
```

Tester si `n` est positif, nul ou négatif

```
n = -7
if n > 0:
    print("positif")
elif n == 0:
    print("nul")
else:
    print("négatif")
```

négatif

ternaire : `x if cond else y`

Même cas que ci-dessus :

```
>>> n = -7
>>> "positif" if n > 0 else "non positif"
'non positif'
```

Ternaire imbriqué (pour gérer les trois cas) :

```
>>> "positif" if n > 0 else ("nul" if n == 0 else "négatif")
'négatif'
```

Boucles

```
for i in range(n):
...
```

Somme des éléments d'une liste

```
L = [4,7,1]
s = 0
for x in L:
    s += x
print(s)
```

12

```
while cond:
...
```

Plus petit diviseur > 1 de `n`

```
n = 15
d = 2
while n % d != 0:
    d += 1
print(d)
```

3

Suites récurrentes

Un pas : $u_{n+1} = \frac{u_n}{2} + 3, u_0 = 1$

```
N = 10 # indices
↪ 0..N-1
u = [0]*N
u[0] = 1
for n in range(N-1):
    u[n+1] = u[n]/2 + 3
print(u)
```

```
[1, 3.5, 4.75, 5.375, 5.6875,
↪ 5.84375, 5.921875,
↪ 5.9609375, 5.98046875,
↪ 5.990234375]
```

Dépendance en `n` : $u_{n+1} = u_n + n, u_0 = 0$

```
N = 10 # indices
↪ 0..N-1
u = [0]*N
u[0] = 0
for n in range(N-1):
    u[n+1] = u[n] + n
print(u)
```

```
[0, 0, 1, 3, 6, 10, 15, 21,
↪ 28, 36]
```

Deux pas : $u_{n+1} = u_n + u_{n-1}, u_0 = 0, u_1 = 1$

```
N = 10 # indices
↪ 0..N-1
u = [0]*N
u[0], u[1] = 0, 1
for n in range(1, N-1):
    u[n+1] = u[n] + u[n-1]
print(u)
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21,
↪ 34]
```

Seuil : plus petit `n` tel que $u_n > 100, u_{n+1} = u_n + 7, u_0 = 0$

```
u = [0] # append
while u[-1] <= 100:
    u.append(u[-1] + 7)
print(u)
print(f"n={len(u)-1}")
```

```
[0, 7, 14, 21, 28, 35, 42, 49,
↪ 56, 63, 70, 77, 84, 91,
↪ 98, 105]
n=15
```

→ `u[n+1] = u[n] + u[n-1]` recopie **littéralement** la formule mathématique; il faut d'abord réserver la liste (`[0]*N`) avant de la remplir
→ taille inconnue à l'avance pour le seuil, donc `u.append(...)` plutôt qu'une liste pré-remplie; `u[-1]` est le **dernier** terme calculé
→ on cherche le plus petit n tel que $u_n > 100$: la boucle **while** doit donc **continuer tant que la condition cherchée n'est pas atteinte**, c'est-à-dire tant que sa **négation** est vraie :
condition cherchée : `u[-1] > 100` → condition de la boucle **while** : `not(u[-1] > 100)` équivaut à `u[-1] <= 100`

Compréhensions

```
>>> [x**2 for x in range(4)]
[0, 1, 4, 9]
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
```

- liste : `[expr for x in iterable if cond]`

Motif fréquent : parcourir les chiffres d'un entier

```
n = 4205
somme = 0
for c in str(n):          # c parcourt les chiffres, en str
    somme += int(c)
print(somme)

11
```

Même chose, en compréhension :

```
>>> n = 4205
>>> sum([int(c) for c in str(n)])
11
```

Produit scalaire de deux listes (zip + compréhension)

```
>>> L1 = [1,2,3]
>>> L2 = [4,5,6]
>>> sum([a*b for a,b in zip(L1,L2)])
32
```

Fonctions (def)

```
def f(a, b=val):
    ...
    return resultat
```

Définir et appeler une fonction

```
def carre(x):
    return x**2
print(carre(3))
```

9

Argument par défaut

```
def puissance(x, n=2):
    return x**n
print(puissance(3), puissance(3,3))
```

9 27

Renvoyer plusieurs valeurs (tuple)

```
def minmax(L):
    return min(L), max(L)
a,b = minmax([3,1,4])
print(a,b)
```

1 4

- variable locale $\neq$ variable globale : une fonction *lit* les variables globales mais ne peut pas les réaffecter (sauf objets mutables modifiés en place).

Fonctions anonymes (lambda)

Déclaration avec lambda (idem def)

<code>f = lambda x,y: 2*x+y</code>	<code>def f(x,y):</code> <code> return 2*x+y</code>
------------------------------------	---

Utilisation (identique dans les deux cas) :

```
>>> f = lambda x,y: 2*x+y
>>> res = f(3,1)
>>> res
7
```

- pas de parenthèses autour des paramètres, **return** implicite, **une seule ligne**
- pratique pour une fonction courte, ex. en compréhension ou passée en paramètre

Importer un module – 4 méthodes

Méthode 1 (générale) :

```
>>> import math
>>> math.sqrt(9)
3.0
```

Méthode 2 (avec alias) :

```
>>> import numpy as np
>>> np.sqrt(9)
3.0
```

Méthode 3 (sélective) :

```
>>> from math import sqrt, pi
>>> sqrt(9)
3.0
```

Méthode 4 ("paresseux", déconseillée) :

```
>>> from math import *
>>> sqrt(9)
3.0
```

→ risque de collision de noms entre modules (ex. `plot` existe dans `matplotlib` et `sympy`)

Module math

```
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.floor(3.7)
3
```

```
>>> math.ceil(3.2)
4
>>> math.factorial(5)
120
```

Attention : comparer des flottants avec == est dangereux

```
>>> 0.1 + 0.2 == 0.3
False
>>> 0.1 + 0.2
0.30000000000000004
>>> math.isclose(0.1 + 0.2, 0.3)
True
```

→ les flottants sont stockés en binaire de façon approchée; on compare deux flottants avec `math.isclose(a,b)`, jamais avec `==`

Graphiques (matplotlib + numpy)

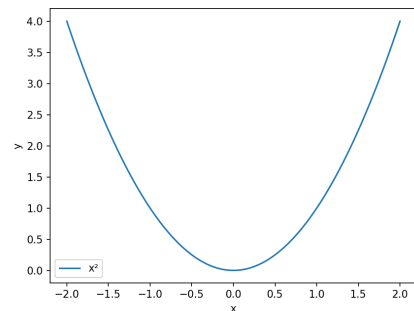
- `import numpy as np` `import matplotlib.pyplot as plt`
- `np.linspace(a,b,n)` : n points régulièrement espacés entre a et b

```
>>> import numpy as np
>>> np.linspace(0,1,5)
array([0. , 0.25, 0.5 , 0.75, 1.  ])
```

- `plt.plot(X, Y)` : trace la courbe reliant les points (x_i, y_i)
- `plt.xlabel("...")` `plt.ylabel("...")`
- `plt.title("...")` `plt.legend()` (nécessite `plt.plot(X,Y,label="...")`)
- `plt.show()` : affiche la figure

Tracer $y = f(x) = x^2$ sur $[-2, 2]$

```
import numpy as np
import matplotlib.pyplot as plt
f = lambda x: x**2
xx = np.linspace(-2,2,100)
yy = f(xx)
plt.plot(xx, yy, label="x²")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.savefig("aide-memoire-plot.png", dpi=150)
plt.show()
```



→ `f` s'applique directement à tout le tableau `xx` (pas besoin de boucle/compréhension) – `plt.savefig(...)` sauvegarde la figure pour l'importer ici même