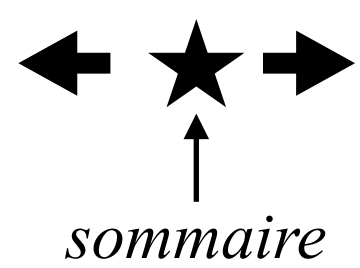
JP. Roy
Réf : "Python, Apprentissage Actif", Ellipses, 2020.

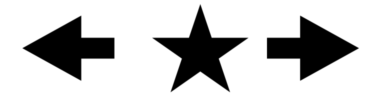


Python en Première

(mémento)

Ce mémento n'est pas un cours structuré ! Il va à l'essentiel.

Le programme Algo-Prog de Première



• Cette année, nous approfondissons quelques notions vues en Seconde et introduisons un nouveau type de données : les listes. Le but visé reste l'acquisition d'une **démarche algorithmique** :

- La notion de **fonction**.
- La **programmation** comme production d'un **texte** destiné à être **exécuté par une machine**. Pour résoudre des problèmes.

Le logiciel Python

• Vous pouvez installer le **logiciel Python** sur votre :

ordinateur → www.python.org ou thonny.org

tablette/smartphone → Pydroid 3 (Androïd) ; Pythonista 3, Pyto, Carnets (iOS)

Non testés…

Partie 1

Révisions de Seconde++
Une panoplie de calculs typiques

Relire le *Mémento de Seconde*…

1. Affectation entre couples, triplets, etc

• Ou **affectation multiple** (ou *déstructurante*).

```
>>> (a,b) = (2,3)
>>> print('a =',a,'et b =',b)
a = 2 et b = 3
```

~~(a,3) = (2,b)~~

tuple de variables = tuple de valeurs
(de même longueur !)

• Réservez *en général* cette affectation multiple au cas où le tuple de droite ne contient que des **valeurs** et non des expressions à calculer.

• **ATTENTION**. Si expr1 et expr2 sont des expressions quelconques :

(a,b) = (expr1,expr2) ~~⟺~~ *en général* | a = expr1 ; b = expr2

Un exemple svp ?

• Philosophie de l'affectation : on évalue à droite puis on transfère à gauche. Vérifiez avec (a,b) = (a + 1,a + b)…

2. Echange de deux variables a et b

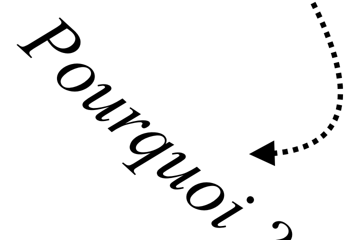
```
>>> (a,b) = (2,3)
```

$\Longrightarrow$

```
>>> (a == 2) and (b == 3)
True
```

- L'algorithme d'échange **faux** :

```
a = b ; b = a
```



Rappel :

expr1 ; expr2 $\Leftrightarrow$ **expr1** / **expr2**

- L'algorithme d'échange **correct**, avec une variable temporaire :

```
tmp = a
a = b
b = tmp
```

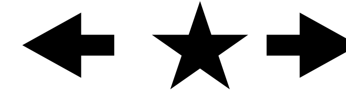
| a == 2, b == 3 | —**tmp = a**→ | a ==, b ==, tmp == | —**a = b**→ | a ==, b ==, tmp == | —**b = tmp**→ | a ==, b == |

- L'algorithme d'échange **correct**, sans variable temporaire (Python) :

```
(a,b) = (b,a)
```

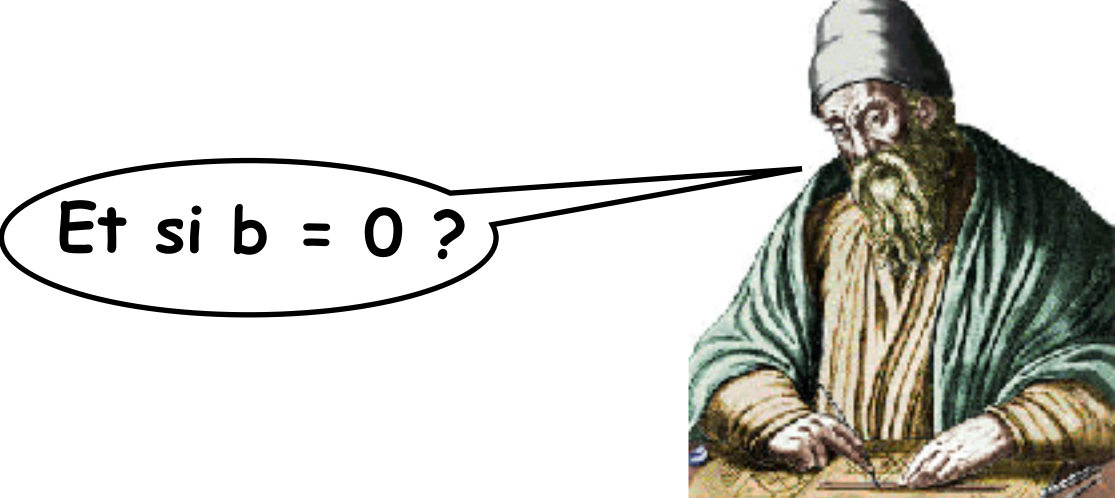
En cas de doute, relire le bas de la page 4…

3. Calcul du PGCD de deux entiers naturels a et b



- Avec l'algorithme d'Euclide.

Le PGCD de a et b est le même que celui de b et du reste de la division de a par b.



- Précisons un peu cela en Python.

  - Le PGCD de a et de 0 est a. *(Ok ?)*

  - Le reste de la division entière est obtenu avec l'opérateur % (*modulo*).

```
def pgcd(a,b):                    # on traduit Euclide mot à mot
    if b == 0: return a
    return pgcd(b,a % b)          # else: est ici sous-entendu
```

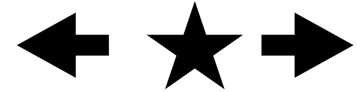
```
>>> pgcd(12,18)
6
```

```
>>> pgcd(0,12)
12
```

```
>>> pgcd(1234,4321)
1
```
premiers entre eux !

- MAIS : UNE FONCTION pgcd QUI S'UTILISE ELLE-MEME ???

Une fonction pgcd qui s'utilise elle-même !



• Oui, et alors ? Pourquoi pas ? Une suite (u_n) définie par récurrence est bien une fonction qui s'utilise elle-même (en maths) :

$$\begin{cases} u_0 = 2 \\ u_{n+1} = 3u_n - 1 \end{cases} \longleftarrow u \text{ utilise } u$$

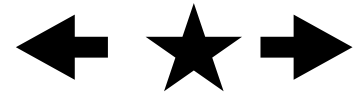
• En programmation, c'est pareil. On peut **programmer par récurrence** ! Et le matheux est content 🙂. Sauf que :

a) Il y a quelques petits soucis techniques avec certains langages de programmation (comme Python) qui n'aiment pas trop la récurrence. 😢

b) De toutes façons : CE N'EST PAS AU PROGRAMME DE 1ère qui met l'accent sur les **boucles**... Donc on transforme le texte par récurrence en une boucle.

```
def pgcd(a,b):          # a et b entiers naturels
    while b != 0:
        (a,b) = (b,a % b)           # cf page 5
    return a
```

Calcul d'une **somme** avec la boucle for (et avec sum)



- Soit à calculer la somme $S = 1^2 + 3^2 + 5^2 + \cdots + 99^2$
- Bon réflexe : on ne reste pas scotché sur ce problème particulier, on rédige une **fonction ré-utilisable**.

```
def somcarimp_v1(a,b):
    '''Retourne la somme des carrés des impairs de [a ; b]
    avec a et b entiers impairs et a ≤ b'''
    res = 0
    for k in range(a,b+1,2):          # de 2 en 2
        res = res + k*k
    return res
```

⇕

```
def somcarimp_v2(a,b):
    return sum(k*k for k in range(a,b+1,2))        # joli ?
```

Une somme en compréhension…

```
>>> somcarimp_v1(1,99)
166650
```

```
>>> somcarimp_v2(1,99)
166650
```

- Le **plus petit diviseur** d'un entier n ≥ 2 :

```
def ppdiv(n):
    '''Retourne le plus petit diviseur ≥ 2 de l'entier n ≥ 2'''
    if n % 2 == 0:          # cas particulier : n est pair
        return 2            # on s'échappe avec le résultat 2
    d = 3                   # (sinon) init. du premier candidat
    while n % d != 0:       # TANT QUE d ne divise pas n:
        d = d + 2           #      je passe à l'impair d suivant
    return d                # on finit avec le résultat d
```

- L'entier n est-il un **nombre premier** ?

```
def est_premier(n):
    return (n >= 2) and (ppdiv(n) == n)
```

- Quel est le **plus grand** nombre premier p à 5 chiffres ?

```
p = 10**5 - 1
while not est_premier(p): p = p - 2
print('p =',p)
```

F5 → p = 99991

La fonction **factorielle** $n \mapsto n!$ sur $\mathbb{N}$

```
def facr(n):     # par récurrence, pour le fun !
    if n == 0: return 1
    return n  * facr(n-1)
```

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)! \end{cases}$$

```
>>> facr(50)          # 50! comporte 65 chiffres…
30414093201713378043612608166064768844377641568960512000000000000
```

```
def fac(n):                          # avec une boucle
    res = 1                          # élément neutre pour *
    for k in range(2,n+1):
        res = res * k
    return res
```

$$n! = \prod_{k=2}^{n} k$$

- Le nombre de chiffres de 5000 ! en utilisant les chaînes :

```
>>> len(str(fac(5000)))
16326
```

et avec facr ? 😢

- Quelle est la première factorielle plus grande que 10^{100} ?

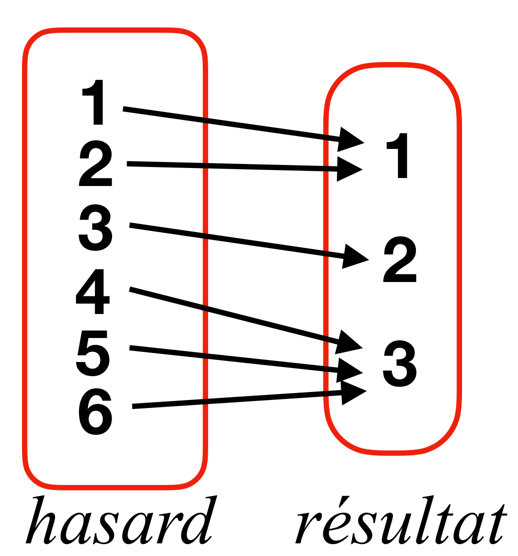
C'est 70! et elle a 101 chiffres. Autre chose, m'sieur ?…

Un dé à 6 faces truqué

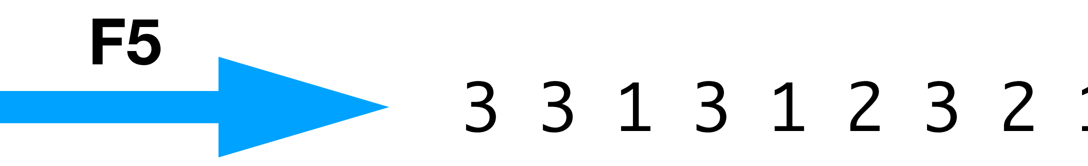
- Construire un dé dont les 6 faces sont 1, 1, 2, 3, 3, 3.

```
from random import randint

def mon_dé():
    hasard = randint(1,6)
    if hasard <= 2: return 1
    if hasard == 3: return 2
    return 3
```




```
for i in range(1,10):
    print(mon_dé(),end=' ')
```



- Vérifions en calculant la probabilité de tirer un 3.

```
nb_tirages = 10000 ; nb_succes = 0
for k in range(nb_tirages):
    if mon_dé() == 3:
        nb_succes = nb_succes + 1
print('Proba(3) =',nb_succes/nb_tirages)
```

F5

Proba(3) = 0.4908

• Construire la **fonction** trinôme $x \mapsto ax^2 + bx + c$ dont les racines sont deux nombres r1 et r2 donnés.

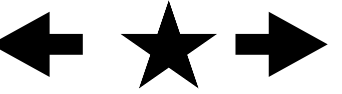
```
def f_trinome(r1,r2):     # float x float → fonction
    '''Retourne une fonction trinôme dont les
    racines sont r1 et r2'''
    return lambda x: (x-r1) * (x-r2)
```

• Problème **inverse**. Sachant qu'une fonction f est bien une fonction trinôme $x \mapsto ax^2 + bx + c$ avec a ≠ 0, trouver les **coefficients** a,b,c.

```
def coeffs(f):    # f est une fonction trinôme
    ....
    return (a,b,c)
```

```
f = f_trinome(-sqrt(2),3)
print(f(-sqrt(2)))        →  -0.0
print(coeffs(f))          →  (1.00…, -1.58…, -4.24…)
```

La dérivation numérique



$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

• En Python, on peut avoir la **fonction dérivée** mais **approchée** :

```
def deriv(f,a,h):       # nombre dérivé approché f'(a)
    return (f(a+h) - f(a)) / h
```

```
>>> f = lambda x: x**3-5*x**2+4
>>> deriv(f,1,0.001)        # ≃ f'(1)
-7.001998999998982
```

• Mais deriv n'est pas propice à calculer f"(1). On fait *abstraction* du paramètre x pour rendre une fonction et non un nombre :

```
def D(f,h):                 # fonction dérivée approchée f'
    return lambda x: (f(x+h) - f(x)) / h
```

```
>>> Df = D(f,0.001)         # f'
>>> D2f = D(Df,0.001)       # f''
>>> D2f(1)                  # f''(1)
-3.9940000000626696
```

Partie 2

Les LISTES

Rappels sur les **séquences**

• Le langage Python permet de programmer sur des objets simples (nombres, booléens) mais aussi sur des *collections* d'objets. En 2nde et 1ère, nous nous limitons aux **séquences** d'objets. Si s est une séquence :

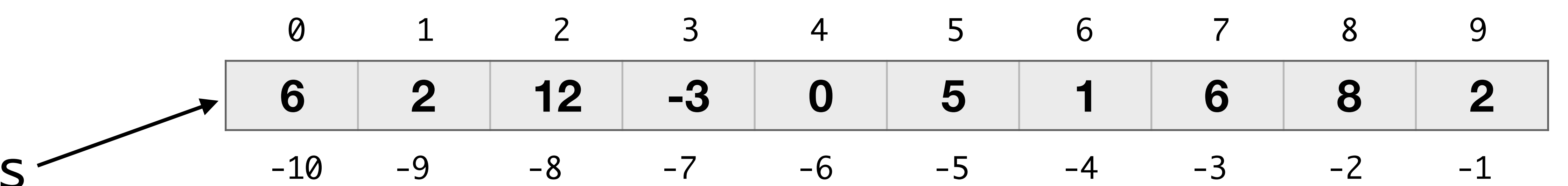
a) On peut compter le **nombre d'éléments** de s (sa longueur) avec l'expression `len(s)` .

b) On peut savoir si x **appartient à** s avec l'expression `x in s`.

c) On peut **boucler** sur les éléments d'une séquence avec l'expression :

```
for x in s: ...
```

d) Les éléments d'une séquence de longueur n sont <u>numérotés</u> de gauche à droite de 0 à n-1. Ou bien de droite à gauche de -1 à -n. L'élément numéro i se note `s[i]`. On y accède **immédiatement** !




```
>>> s[3] == s[-7]
True
```

• Exemples de séquences : `(3,-1,5)` **tuple** `'Azerty'` **str** `range(2,8)` **range**

• Les listes ressemblent aux tuples :

`t = (3,-2,1,'Hello')`

*un **tuple** de longueur 4*

`L = [3,-2,1,'Hello']`

*une **liste** de longueur 4*

```
>>> type(t)
<class 'tuple'>
```

```
>>> type(L)
<class 'list'>
```

• MAIS : les **tuples** ne sont **pas mutables**. Impossible de changer l'élément numéro 3 du tuple `t` sans construire un nouveau tuple !

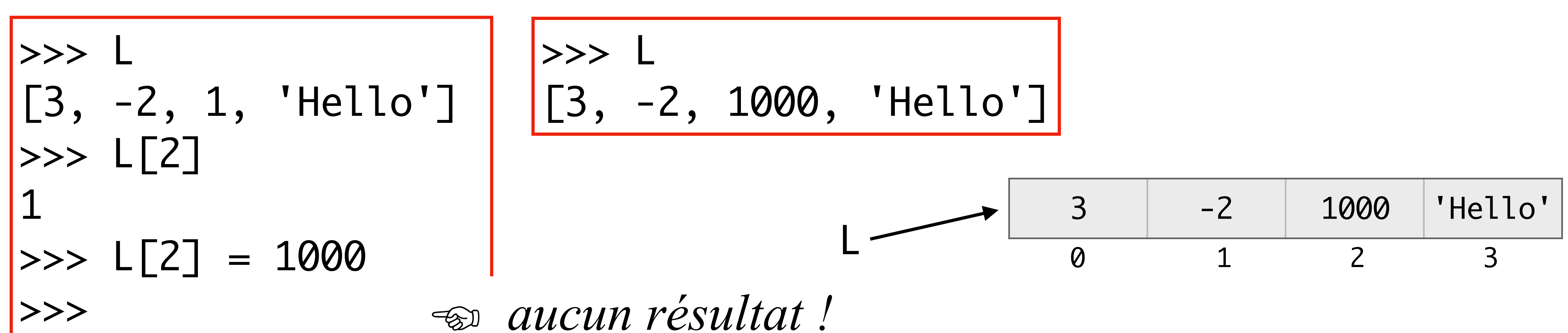
```
>>> t
(3, -2, 1, 'Hello')
>>> t[2]
1
>>> t[2] = 1000
```
Error

```
>>> t
(3, -2, 1, 'Hello')
>>> t = (t[0],t[1],1000,t[3])
>>> t
(3, -2, 1000, 'Hello')
```
●

NB. Un nouveau tuple ● a été créé en mémoire, et `t` *prend comme valeur ce nouveau tuple. L'ancien tuple (3,-2,1,'Hello') sera détruit, plus tard...*

Les **listes** sont des séquences **mutables** !

• Il est possible de **modifier** l'élément numéro 2 de la liste L **sur place** : sans avoir besoin de construire une nouvelle liste !



NB. a) ***Très efficace*** *! Aucune construction de nouvelle liste en mémoire. On ouvre un tiroir dont on change le contenu, sans changer le tiroir…*
b) Ouvrez le Python Tutor et essayez les exemples des pages 15 et 16.

• Il est parfois utile d'obtenir une **tranche** de liste (*slice*) :

```
>>> L = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> L[3:8]                         # indices dans [3,8[
[8, 10, 12, 14, 16]                # copie d'une tranche de L
```

Comment **construire** une liste ?

- En **extension**, en donnant tous ses éléments :

```
>>> L = [3, -2, 1, 'Hello']
>>> len(L)          # longueur
4
```

```
>>> L1 = []
>>> len(L1)
0
```

- En **compréhension**, en donnant la manière de calculer chaque élément :

```
>>> [x for x in range(1,11)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [x for x in range(1,11) if x % 2 == 1]
[1, 3, 5, 7, 9]
>>> [x for x in L if (type(x) == int) and (x > 0)]
[3, 1]
```

- En transformant une autre séquence en **liste** :

```
>>> list(range(0,5))
[0, 1, 2, 3, 4]
>>> list('azerty')
['a', 'z', 'e', 'r', 't', 'y']
```

```
>>> t = (3,7,8,-1)
>>> list(t)
[3, 7, 8, -1]
```

- En **concaténant** deux listes :

```
>>> [1, 3, 5] + [2, 4, 3]
[1, 3, 5, 2, 4, 3]
```

☜ *une nouvelle liste en mémoire…*

- En **programmant** soi-même une **fonction** qui retourne une **liste** :

```
def diviseurs(n):
    '''Retourne la liste des diviseurs de l'entier n'''
    res = []
    for d in range(1,n+1):
        if n % d == 0:
            res = res + [d]
    return res
```

```
>>> diviseurs(50)
[1, 2, 5, 10, 25, 50]
```

NB. A chaque diviseur, on construit une nouvelle liste en mémoire. Cette manière de programmer n'est pas très efficace (cf page 19) !

```
def diviseurs(n):
    return [d for d in range(1,n+1) if n % d == 0]
```

Une liste en compréhension est souvent suffisante… et efficace.

Comment ajouter un élément (à droite !) à une liste L ?

- La **mauvaise** manière (en général) : L + [x]

```
>>> [1, 3, 5, 7] + [9]
[1, 3, 5, 7, 9]
```

☞ *construit une nouvelle liste à 5 éléments (coût = 5)*

- La **bonne** manière (en général) : **L.append(x)**

```
>>> L = [1, 3, 5, 7]
>>> L.append(9)          # aucun résultat !
>>> L
[1, 3, 5, 7, 9]          # coût = 1 (sur place)
```

- Reprenons la fonction diviseurs de la page précédente :

```
def diviseurs(n):
    '''Retourne la liste des diviseurs de l'entier n'''
    res = []
    for d in range(1,n+1):
        if n % d == 0:
            res.append(d)    # ajouter d à droite
    return res
```

C'est ce que fait la compréhension de liste !

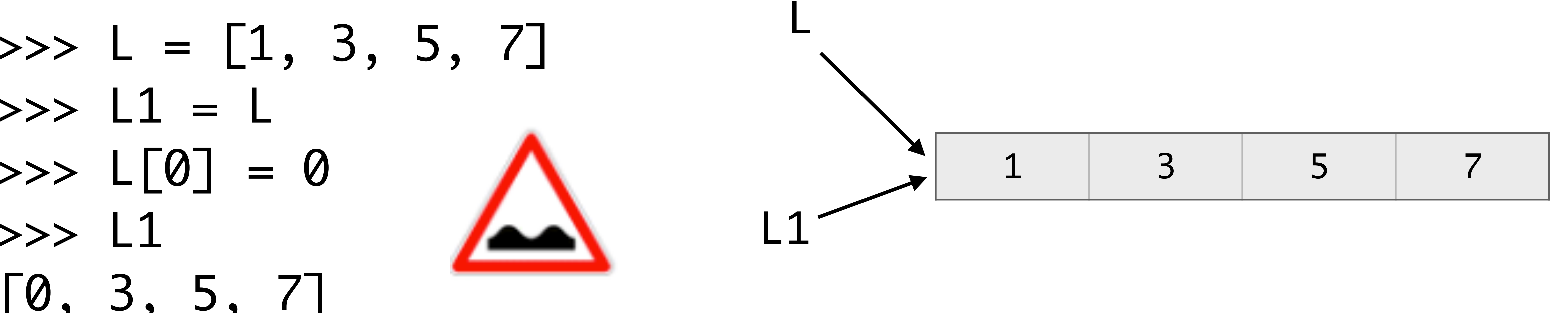
Comment ajouter une liste d'éléments à une liste L ?

- Avec une variante de .append qui se nomme .extend :

```
>>> L = [1, 3, 5, 7]
>>> L.extend([9,11,13])          # aucun résultat !
>>> L
[1, 3, 5, 7, 9, 11, 13]
```

UNE PETITE MISE EN GARDE SUR LA MUTATION DES LISTES

- L'instruction d'affectation L1 = L ne construit **aucune nouvelle liste** en mémoire. Elle se contente de dire que L est un synonyme (un **alias**) de L.



On dit que L et L1 partagent une zone mémoire...

• **Module** très complet pour **dessiner en maths**. Considérons le tracé de la courbe d'une fonction f : $[a,b] \to \mathbb{R}$.

```
f = lambda x: 2*x*cos(x/4)
```

$\Longleftrightarrow$

```
def f(x):
    return 2*x*cos(x/4)
```

• La courbe étant approchée par une **suite de petits segments** reliant les points de la courbe, il suffit en fait de disposer d'un tableau de points de la fonction f, formé d'une **liste d'abscisses** Lx et d'une **liste d'ordonnées** correspondantes Ly.

• La liste des abscisses Lx est obtenue en divisant l'intervalle [a,b] en N points de même espacement, avec N assez grand. La fonction ci-dessous **linspace(a,b,N)** retourne une telle liste :

```
def linspace(a,b,N):
    '''Retourne une liste de N points x0=a,x1,…,xN-1=b'''
    h = (b-a) / (N-1)          # l'espacement entre deux points
    return [a+i*h for i in range(0,N)]     # en compréhension…
```

• Exemple d'utilisation de `linspace` :

```
>>> linspace(0,1,11)          # 11 points, 10 sous-intervalles
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
```

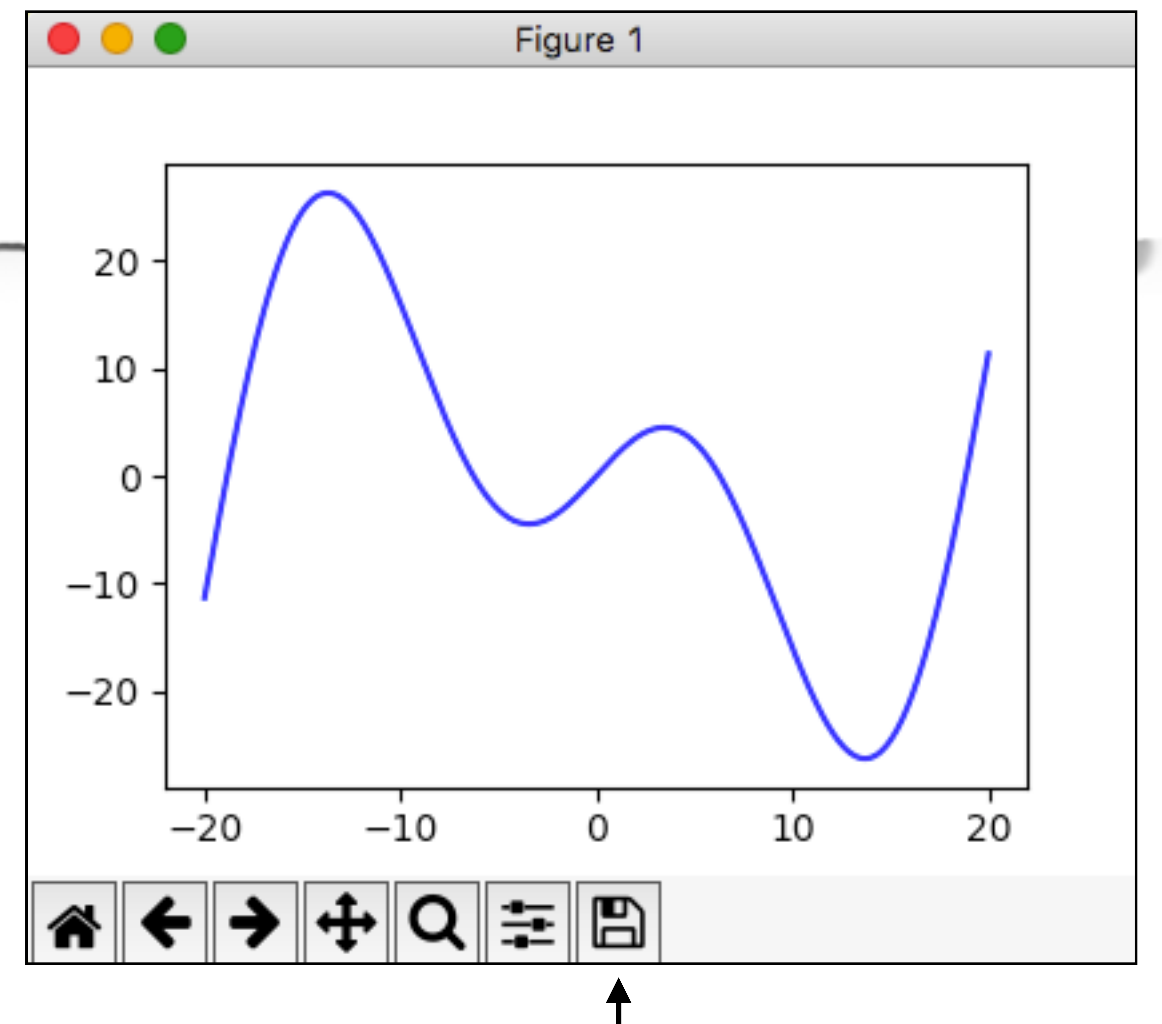
• Le **code pour tracer une courbe** sera en général une variante de :

```
from matplotlib import pyplot as plt
f = lambda x: 2*x*cos(x/4)
Lx = linspace(-20,20,300)    # découpage de [-20,20] en 300 points
Ly = [f(x) for x in Lx]
plt.plot(Lx,Ly,'b-')    # calcul
plt.show()              # affichage
```

• `'b-'` pour des segments bleus.

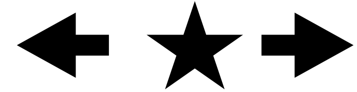
• *La fonction* `linspace` *existe en Python, mais dans un module* **numpy** *destiné aux étudiants en Science, et un peu compliqué...*

```
import numpy as np
Lx = np.linspace(-20,20,300)
```



sauvegarder l'image xxx.png

Dessiner un nuage de points avec le module matplotlib



• Pour dessiner un nuage de points (x_n,y_n), par exemple issu d'une suite, ou d'une série statistique, on peut utiliser même technique Lx, Ly.

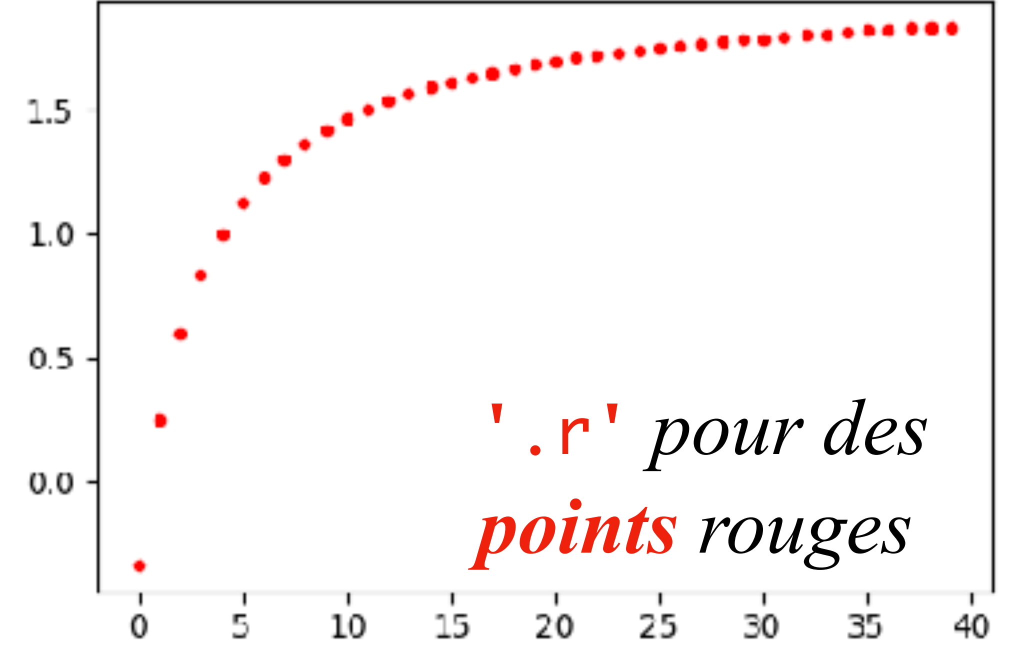
```
u = lambda n: (2*n-1)/(n+3)
```

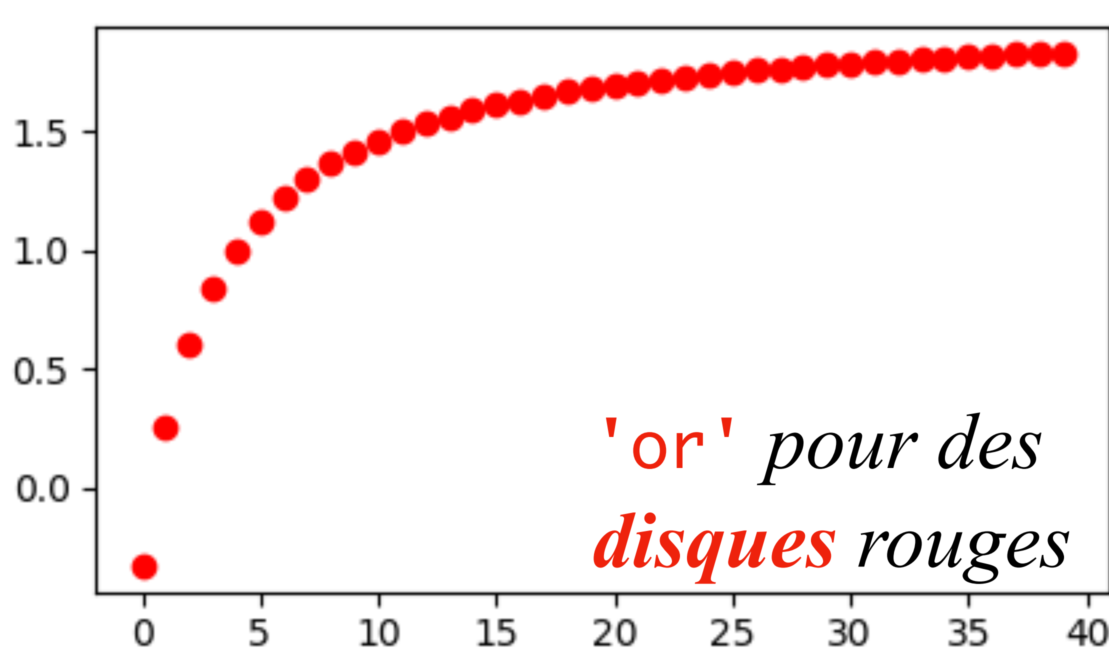
⟺

```
def u(n):        # n entier ≥ 0
    return (2*n-1)/(n+3)
```

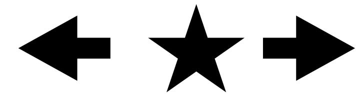
```
from matplotlib import pyplot as plt
Lx = [n for n in range(0,40)]      # ⟺ list(range(0,40))
Ly = [u(n) for n in Lx]
plt.plot(Lx,Ly,'or')
plt.show()
```

On peut aussi utiliser la fonction `plt.scatter`.





Dessiner une **suite définie par récurrence** $u_{n+1} = f(u_n)$



• Pour du travail avec des points du plan, notamment les suites définies par récurrence. Par exemple : $u_0 = -1/2, \quad u_{n+1} = f(u_n)$ avec :

```
def f(x):        # x ≥ -1
    return sqrt(x+1)
```

$$f(x) = \sqrt{x+1}$$

```
# calcul de la liste Lu = [u0,u1,u2,u3,u4,u5,u6,u7]
Lu = [-1/2]
for n in range(1,7):
    Lu.append(f(Lu[-1]))   # L[-1] est le dernier terme calculé
print('Lu =',Lu)
```

```
Lu = [-0.5,0.70…,1.30…,1.51…,1.58…,1.60…,1.61…,1.61…]
```

```
# calcul du graphique (mise en place)
from matplotlib import pyplot as plt
plt.title('Suite définie par récurrence u(n+1)=f(u(n))')
plt.axis('equal')                     # repère orthonormé !
```

```
# affichage des axes en noir (k). Deux points sur chaque axe
plt.plot([-1,2],[0,0],'-k')    # axe Ox, de (-1,0) jusqu'à (2,0)
plt.plot([0,0],[0,2],'-k')     # axe Oy, de (0,0) jusqu'à (0,2)
```

```
# affichage des points u0,u1,u2,u3 en abscisses, en noir
for n in range(0,4):
    plt.plot([Lu[n]],[0],'ok')              # un disque noir
    plt.text(Lu[n]-0.05,-0.15,'u'+str(n))
```

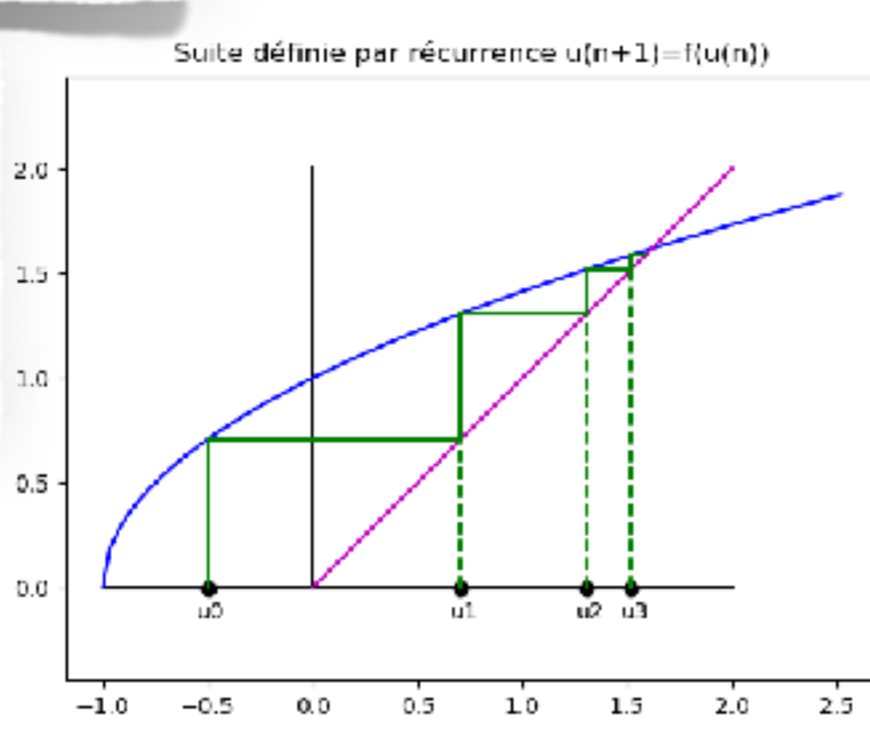
```
# affichage de la courbe de f en bleu sur [-1 ; u3+1]
Lx = linspace(-1,Lu[3]+1,100)
Ly = [f(x) for x in Lx]
plt.plot(Lx,Ly,'-b')
```

```
# affichage de la droite y = x en magenta
plt.plot([0,2],[0,2],'-m')    # de (0,0) jusqu'à (2,2)
```

```
# affichage des segments verticaux, cas particulier de u0
plt.plot([Lu[0],Lu[0]],[0,Lu[1]],'-g')          # en vert
plt.plot([Lu[0],Lu[1]],[Lu[1],Lu[1]],'-g')
```

```
# affichage des segments verticaux, cas général
for n in range(1,4):
    plt.plot([Lu[n],Lu[n]],[Lu[n],Lu[n+1]],'-g')
    plt.plot([Lu[n],Lu[n]],[0,Lu[n]],'--g')
    plt.plot([Lu[n],Lu[n+1]],[Lu[n+1],Lu[n+1]],'-g')
```

```
plt.show()
```



'--g' pour pointillés en vert

SOMMAIRE

Page		Page	
1	Titre	15	Les séquences
2	Le programme de Première	16	Les listes
3	Partie 1	17	Les listes sont mutables
4	Affectations entre tuples	18	Construction d'une liste (1)
5	Echange de deux variables	19	Construction d'une liste (2)
6	Le PGCD de deux entiers naturels	20	Ajouter un élément à une liste
7	Une fonction définie par récurrence	21	Mise en garde sur la mutation
8	Calculer avec for	22	Dessiner la courbe d'une fonction (1)
9	Calculer avec while	23	Dessiner la courbe d'une fonction (2)
10	La factorielle n!	24	Dessiner un nuage de points
11	Un dé truqué	25	Dessiner une suite u(n+1)=f(u(n)) (1)
12	Construction dynamique d'une fonction	26	Dessiner une suite u(n+1)=f(u(n)) (2)
13	Fonction dérivée	27	SOMMAIRE
14	Partie 2		

matplotlib