Python au lycée pour les profs

- Saison 2 -

Le programme Algo-Prog de la Saison 2 (1ère+)

• Nous approfondissons les notions vues dans la saison 1 et introduisons un nouveau type de données : les **listes**. Le but visé reste l'acquisition d'une première **démarche algorithmique** à travers :

- la notion de **fonction**
- la **programmation** comme rédaction d'un **texte** destiné à être **exécuté par une machine**. Dans le but de résoudre des problèmes.

problem-solving...

Le logiciel Python

• Vous pouvez installer le logiciel **Python 3** sur votre :

ordinateur → www.python.org ou Anaconda...

tablette/smartphone → Pydroid 3 (Android), Pythonista 3 (iPad), Carnets (iPad)

Les objets utilisés par Python (revoir la saison 1)

• Chaque ***objet*** appartient à une ***classe*** (ou ***type***) qui définit les *opérations de base* légales sur ses objets. Par exemple, 352 est un objet de la classe `int`.

• Les **nombres**, types `int` (entiers exacts) et `float` (nombres approchés).

• Les **textes**, type `str` (chaînes de caractères).

• Les **collections** d'objets, types `str` (chaînes) et `tuple` (couples, triplets...).

• La **tortue** (une seule tortue sans nom, ou plusieurs tortues nommées).

```
157706340        3.141592653589793        'Thx-38 !'    (1.5,-8)        
```
`int` `float` `str` `tuple` `Turtle`

Et quel nouveaux acteurs dans la saison 2 ?

• Quelques compléments sur les *tuples* et les *chaînes de caractères*.

• Une nouvelle collection, celle des **listes** (type `list`) qui ressemblent aux tuples mais dont on peut **modifier** les contenus. Des acteurs majeurs !

Le minimum à savoir pour la saison 2

• Les **entiers relatifs** (type `int`) sont **exacts**. Les deux opérations importantes sont le **quotient** `a // b` et le **reste** de la division `a % b`. Le PGCD se nomme `gcd` et se trouve dans le module `math`.

```
from math import gcd, lcm
```

Greatest Common Divisor, Lowest Common Multiple

• Les **nombres ~~réels~~ flottants** (type `float`) sont **approchés** donc *inexacts* (l'égalité == est proscrite). La division approchée se note /. La notation est `3.14159` ou bien en virgule flottante `314.159e-2` (`e-2` pour 10^{-2}). La plupart des fonctions mathématiques usuelles sont dans le **module** `math`, il faut les importer si besoin. Seule `abs` est dans le noyau Python.

```
from math import sqrt, sin, cos, pi
```

```
from math import *
```

déconseillé !

• Un **couple** se note `(a,b)` en Python, un **triplet** `(a,b,c)` etc. Les composantes d'un couple `c` se notent `c[0]`, `c[1]`, etc. Un couple est de type `tuple`. On ne peut **pas modifier** les composantes d'un tuple.

Attention : `(1,5) + (8,2) == (1,5,8,2)` ⟵ *une "concaténation"*

• Python peut calculer sur des **chaînes de caractères** (*textes*, de type `str`). La longueur d'une chaîne `s` se note `len(s)`. Les **caractères** notés `s[k]` sont numérotés avec $k \in$ [0 ; len(s)-1]. Une chaîne se note `'Comme ceci !'` ou `"Comme cela !"`, ou encore `"Jusqu'à midi"`. On peut **modifier** le caractère numéro k d'une chaîne avec `s[k] = x`.

Attention : `'Calculer ' + '2*3' == 'Calculer 2*3'` ~~`2 + '3158'`~~ *mauvais typage*

• La rédaction d'un programme Python passe par le codage de **fonctions** recevant des **données** et produisant un **effet** ou un **résultat**.

sans résultat, avec un effet

```
def div1(a,b) :
    q = a // b
    r = a % b
    print('div1 :',q,r)

div1(22,5)
```

```
div1 : 4, 2
```

Le résultat est None*... qui ne s'affiche pas.*

avec résultat, sans effet

```
def div2(a,b) :
    q = a // b
    r = a % b
    return (q,r)

print('div2 :',div2(22,5))
```

MIEUX !

```
div2 : (4, 2)
```

Le résultat est ici un couple.

• Souvent, une fonction utilise une boucle (**while** ou **for**) qui exécute la même suite d'instructions jusqu'à obtention du résultat ou de l'effet désiré.
Il faut **initialiser les variables de boucle**.

• Dans une boucle `while`, la répétition utilise un **test**.

while <continue?> :
    instruction$_1$
    ...
    instruction$_n$

```
def truc(a,b) :
    q = 0      # initialisation
    while a >= b :
        a = a - b
        q = q + 1
    return q
```

```
truc(32,7)
    = ?
```

• Dans une boucle `for`, le nombre de répétitions est connu à l'avance avec `range`.

for i in range(a,b) :
    instruction$_1$
    ...
    instruction$_n$

```
def machin(a,b) :
    s = 0      # initialisation
    for i in range(a,b+1) :
        s = s + i*i
    return s
```

```
machin(3,10)
    = ?
```

range(n) ⇝ **0,1, …, n-1** **range(p,q)** ⇝ **p, p+1, … , q-1** **range(p,q,s) : range(p,q) de s en s**

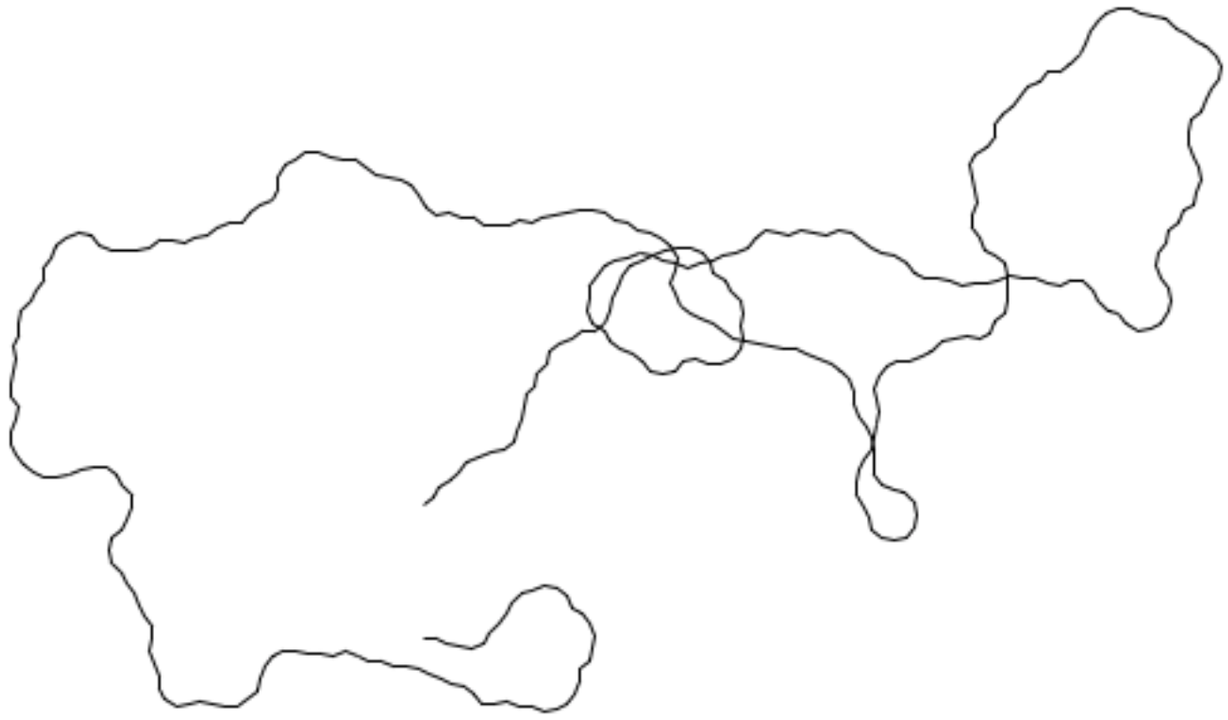
• Pour générer un **entier** aléatoire dans [a ; b] : `randint(a,b)`. Le dé bien équilibré à 6 faces : `randint(1,6)`. Dans le module `random`.

• Pour générer un **flottant** aléatoire dans]a ; b[: `uniform(a,b)`.

• Les fonctions pour piloter la **tortue** sont dans le module `turtle`.

*Exemple : une **marche aléatoire** de n pas de longueur d, avec une variation maximum dc du cap courant à chaque pas. Résultat : la position finale.*

```
from turtle import *
from random import uniform

def marche_alea(n,d,dc) :
    clearscreen()
    ht()
    setheading(uniform(0,360))
    for i in range(n) :
        forward(d)
        left(uniform(-dc,dc))
    return pos()

M = marche_alea(300,5,40)
print('Position finale', M)
```



```
Position finale (-0.21,52.95)
```

https://perso.limsi.fr/pointal/_media/python:turtle:turtleref.pdf

• La fonction print(x,y,…) affiche les valeurs des expressions x,y . . .

✳ en les **séparant** par un espace ✳ en **terminant** par un saut de ligne

print(x,y,…) ⟺ print(x,y,…,sep=' ',end='\n')

• On peut modifier ce comportement *par défaut* en précisant d'autres valeurs pour les chaînes sep et end.

```
for i in range(5) :
    print(i)
print('FINI')
```

```
0
1
2
3
4
FINI
```

```
for j in range(2,15,3) :
    print(j,end='.')
print('\nFINI')
```

```
2.5.8.11.14.
FINI
```

end='' *pour ne rien faire en fin de ligne*

```
print(1,2,3,sep='-')
```

```
1-2-3
```

```
print(1,2,3,end='-')
```

```
1 2 3-
```

• Il est IMPORTANT de maîtriser le fonctionnement de range, sep et end.

• Méthodologie au niveau du programme. Décomposez le problème en problèmes **plus simples**, de quelles **fonctions** doit-on disposer pour le résoudre ? Précisez le cahier des charges de chaque fonction : ses arguments (leur type ?), son résultat (s'il y en a plusieurs : un tuple).

• Méthodologie au niveau d'une fonction. Essayez d'abord de l'obtenir en composant d'autres fonctions (existantes ou non). Sinon réfléchissez au moyen de passer de la donnée au résultat avec une **boucle** ; de quelles variables aurez-vous besoin ? L'idéal : disposer d'une assertion qui relie toutes ces variables de boucle et qui soit vérifiée à chaque tour ★ de boucle : *délicat au lycée…*

```
def division(a,b) :
    q = 0
    r = a
    while ★ r >= b :
        r = r - b
        q = q + 1
    return (q,r)
```

a = 22, b = 5

```
>>> division(22,5)
(4, 2)
```

*La relation **a=bq+r** est vérifiée à chaque tour de boucle.*

q	r
0	22
1	17
2	12
3	7
4	2

En sortie, on a en plus $r < b$.

• Une **fonction** Python renvoie toujours **un seul résultat** ! La fonction précédente `division(a,b)` retourne deux objets mais regroupés dans un seul tuple.

$$\texttt{division} : \mathbb{N} \times \mathbb{N}^* \longrightarrow \textit{tuple}$$

• Mais comment utiliser le tuple résultat `(4,2)` ?

```
>>> d = division(22,5)
>>> d
(4, 2)
>>> q = d[0]
>>> r = d[1]
```

Hum...

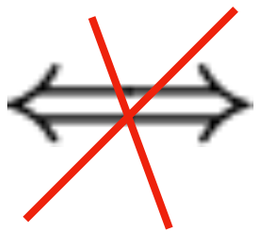
mieux

```
>>> (q,r) = division(22,5)
```

parce que l'on SAIT que le résultat est un couple !

• **Attention** néanmoins à l'**affectation d'un couple**, **triplet**, etc.

```
(x,y) = (expr1,expr2)
```



```
x = expr1
y = expr2
```

expr1 et expr2 sont calculées *AVANT* que l'affectation soit faite !

```
(x,y) = (3,2)
```

⟶

```
(x,y) = (x+y,x-y)
```

⟶

```
x = 5
y = 1
```

et non
x=5
y= 3

Le point de vue d'une IA (sur poe.com) <<<<<<<<<<<<<<<<<<<<<<<<<<

En Python, les **entiers** et les **flottants** sont deux types de données utilisés pour représenter des nombres, mais ils ont des caractéristiques différentes.

Entiers (int)

Les **entiers** sont des nombres sans décimales. Ils peuvent être positifs, négatifs ou nuls. Par exemple, -5, 0 et 42 sont des entiers. En Python, il n'y a **pas de limite à la taille d'un entier**, ce qui signifie que tu peux travailler avec des nombres très grands, tant que tu as suffisamment de mémoire.

Flottants (float)

Les **flottants**, en revanche, représentent des nombres réels avec des décimales. Par exemple, 3.14, -0.001 et 2.0 sont des flottants. Cependant, en raison de la façon dont les ordinateurs stockent ces nombres, tous les flottants **ne peuvent pas** être représentés avec une **précision** parfaite. Cela peut conduire à des erreurs lors de calculs.

Notation scientifique

La notation scientifique est une façon d'écrire des nombres très grands ou très petits de manière concise. Par exemple, 1.5×10^3 peut être écrit en Python comme **1.5e3**, ce qui équivaut à 1500. Cela permet de gérer plus facilement des nombres qui auraient autrement beaucoup de chiffres.

Limites des types

Il est important de noter que tous les entiers et flottants ne peuvent pas être représentés en Python. Les entiers peuvent devenir très grands, mais ils nécessitent beaucoup de mémoire. Les flottants, à cause de leur représentation en binaire, peuvent ne pas être capables de représenter certaines valeurs avec précision, ce qui peut affecter les résultats de calculs délicats.

En résumé, les entiers sont de simples nombres sans décimales, tandis que les flottants peuvent inclure des décimales, mais avec des limitations de précision.

➡ ***J'ajouterais : ne PAS utiliser == avec des nombres flottants !***

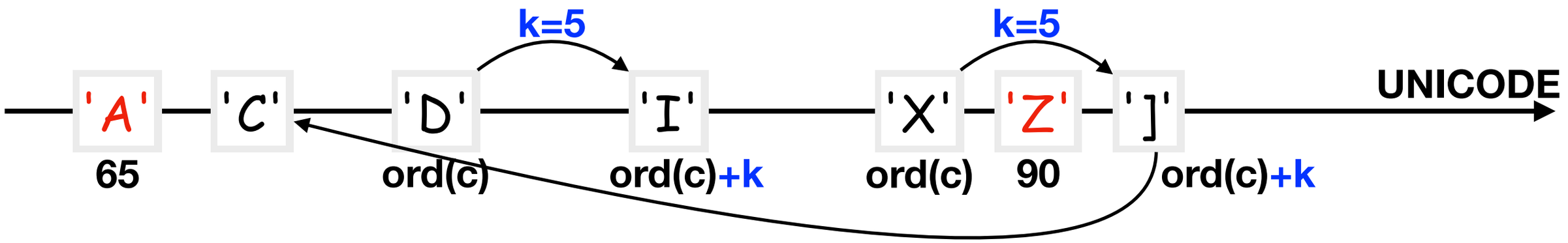
POUR ÊTRE SÛR D'AVOIR BIEN RÉVISÉ…

- Donnez un exemple de collection en Python.
- Donnez deux écritures différentes en virgule flottante de 0.5317.
- Pourquoi vaut-il mieux coder des fonctions avec résultat ?
- Programmez la fonction exposant(p,n) renvoyant l'exposant du nombre premier p dans la décomposition en facteurs premiers de $n \in \mathbb{N}$.
- Programmez la fonction fac(n) renvoyant n! = 1*2*3*…*n pour $n \in \mathbb{N}$
- ASTUCE : par combien de 0 le nombre 50000! se termine-t-il ?
- Calculez le plus petit entier k > 0 tel que : $1+\frac{1}{2}+\frac{1}{3}+\cdots\frac{1}{k}>5$ $\rightarrow 12499$
- Si l'opérateur ** n'existait pas dans le type int, comment pourriez-vous définir la fonction puissance(x,n) renvoyant x^n avec $n \in \mathbb{N}$?
- Comment échanger les valeurs des variables a et b ? *Plusieurs solutions.*
- Avec la tortue, dessinez un cercle centré en (0,0) et de rayon 200. Rédigez un code produisant une marche aléatoire de la tortue de telle sorte que la tortue partant de (0,0) reste dans le disque…

UNICODE contient tous les **caractères** du monde (langues, maths, musique, braille, etc) numérotés. Ex: chr(65) == 'A', ord('A') == 65.
Les chiffres '0'…'9' se suivent, ainsi que les lettres 'A'…'Z' et 'a'…'z'.

```
chr(68) == ?          ord('Y') == ?          ord(' ') == 32
```

• Voici le codage secret de **César** pendant la guerre des Gaules. Nous supposons que s est un mot en *majuscules non accentuées* ! La **clé** k sera un entier de [1 ; 25].




```
def cesar(s,k) :      # renvoie une copie de s codée par la clé k
    res = ''                        # le résultat, vide au départ
    for c in s :                    # pour chaque caractère c de s
        num = ord(c) + k            # unicode de c poussé de k
        if num > ord('Z') :         # si débordement à droite,
            num = num - 26          #    on recule
        res = res + chr(num)
    return res
```

```
>>> cesar('ATTAQUEZ',5)
'FYYFVZJE'
```

• Voilà donc une nouvelle forme de la boucle `for` qui n'utilise pas range.

L'opérateur `s1 in s2` teste ici si s1 est une *sous-chaîne* de s2.

```
>>> 'b' in 'aeiouy'
False
```

```
>>> 'eio' in 'aeiouy'
True
```

```
>>> 'ioe' in 'aeiouy'
False
```

• En fait, l'opérateur `in` fonctionne sur toutes les **collections** d'objets de Python.

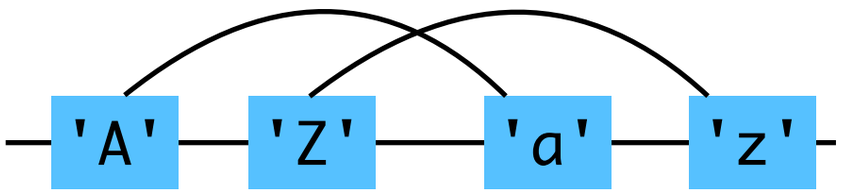
Pour nous : les tuples, les chaînes, et plus tard les listes. Même range(n) peut être considéré comme une collection (potentielle)…

```
>>> 'i' in 'aeiouy'
True
```
∈

```
>>> 3 not in (5,3,2)
False
```
∉

```
>>> 3 in (5,3,2)
True
```

• Le codage UNICODE de tous les caractères permet d'effectuer des opérations diverses sur les textes. Exemple : convertir un caractère **minuscule en majuscule** :




```
>>> decal = ord('a') - ord('A')
>>> chr(ord('r') - decal)
'R'
```
⇝ 32

• Est-il possible d'incorporer des **variables dans une chaîne** ? Oui, en utilisant une **chaîne formatée** *(f-string)* préfixée par la lettre f :

```
>>> n = 3
>>> nom = 'Suzuki'
```

```
>>> f'Ce moteur {nom} a {n} ans !'
'Ce moteur Suzuki a 3 ans !'
```

Ce qui est entre accolades est évalué, comme le serait {2+3}.

NB. Dans les anciennes version de Python (donc souvent dans les livres), il était usuel d'écrire plutôt :

```
'Ce moteur {} a {} ans !'.format(nom,n)
```

• A vous de choisir ce que vous préférez, mais évitez l'ancien .format(…)

```
print('√2 vaut',sqrt(2))   ⟺   print(f'√2 vaut {sqrt(2)}')
```

• La fonction len renvoie le nombre d'éléments d'une collection (chaîne, tuple, liste). Elle permet d'obtenir le nombre de chiffres d'un entier transformé en chaîne :

```
>>> (345, str(345), int('345'))
(345, '345', 345)
```

```
>>> len(str(123456**34))
174                    # 174 chiffres !
```

L'écriture **binaire** d'un entier naturel

• Nous manipulons en général des entiers écrits en **décimal** (base 10) avec des chiffres dans [0 ; 9] comme $\underline{3258} = \underline{8} + \underline{5}\text{x}10^1 + \underline{2}\text{x}10^2 + \underline{3}\text{x}10^3$.

• Si l'on remplace 10 par 2, on passe dans la numération **binaire** dont les chiffres sont 0 et 1. Par exemple en base 2, l'entier 23 s'écrit 10111, vérifiez-le. Ne lisez pas *"dix mille cent onze"* le nombre binaire 10111, prononcez tous ses chiffres…

• Python note 0b10111 l'entier 23 en binaire. Les deux écritures sont équivalentes !

```
>>> 0b10111 + 3
26
```

```
>>> 0b10111 + 0b11
26
```

```
>>> bin(0b10111 + 0b11)
'0b11010'      # une chaîne !
```

• Comment faire afficher les chiffres décimaux (ou binaires) d'un entier n ? Il faut utiliser les propriétés fondamentales de la division par 10 (ou par 2) :

`n % 10` est le chiffre des unités en décimal
`n // 10` est le nombre n privé de son chiffre des unités

```
n = 3258
n % 10 == 8
n // 10 == 325
```

abcde

`n % 2` est le chiffre des unités en binaire
`n // 2` est le nombre n privé de son chiffre des unités

```
p = 0b11010
p % 2 == 0   # pair !
p // 2 == 0b1101
```

• En utilisant la propriété précédente, nous sommes capables de trouver un **algorithme** pour afficher les **chiffres de n** (en décimal ou binaire).
Il suffit de faire une suite de divisions par la base b (10 ou 2) :

```
def aff_chiffres(n,b) :     # affichage des chiffres de n en base b
    while n > 0 :
        print(n % b,end='')
        n = n // b
    print()
```

```
>>> aff_chiffres(3258,10)
8523
>>> aff_chiffres(26,2)
01011
```

26 == 0b11010

• Oui, on obtient les chiffres à l'envers en partant du chiffre des unités, qui est le seul point d'entrée naturel dans un entier… Mieux, nous aurions pu les **stocker dans une collection** (une chaîne ou un tuple) en faisant une fonction **avec résultat** :

```
def chiffres(n,b) :         # la chaîne des chiffres de n en base b
    res = ''
    while n > 0 :
        res = str(n % b) + res
        n = n // b
    return res
```

C'est l'algo de bin !

```
>>> chiffres(3258,10)
'3258'
>>> chiffres(26,2)
'11010'
```

bin(26) == ?

• Attention au *bug* fréquent : `res = res + str(n % b)`. Surveillez l'ordre !!

Les fonctions et les méthodes en Python

• Python n'utilise pas que des **fonctions** (avec ou sans résultat) agissant sur des données, comme :

```
>>> forward(10)      # sans
>>> print(pi/3)      # sans
0.8660254037844386
>>> sin(pi/3)        # avec
0.8660254037844386
```

• On trouve aussi une autre manière de parler, inconnue des maths : l'**envoi de message à un objet**. Par exemple, dans la classe `str`, on peut construire une copie en majuscule de la chaîne `s` en évaluant `s.upper()` alors qu'on s'attendait à écrire simplement `upper(s)`. On dit que `upper` est une **méthode**.

```
>>> s = 'Écrivez ça'
```

```
>>> s.upper()
'ÉCRIVEZ ÇA'
```

```
>>> upper(s)    # oups !
name 'upper' is not defined
```

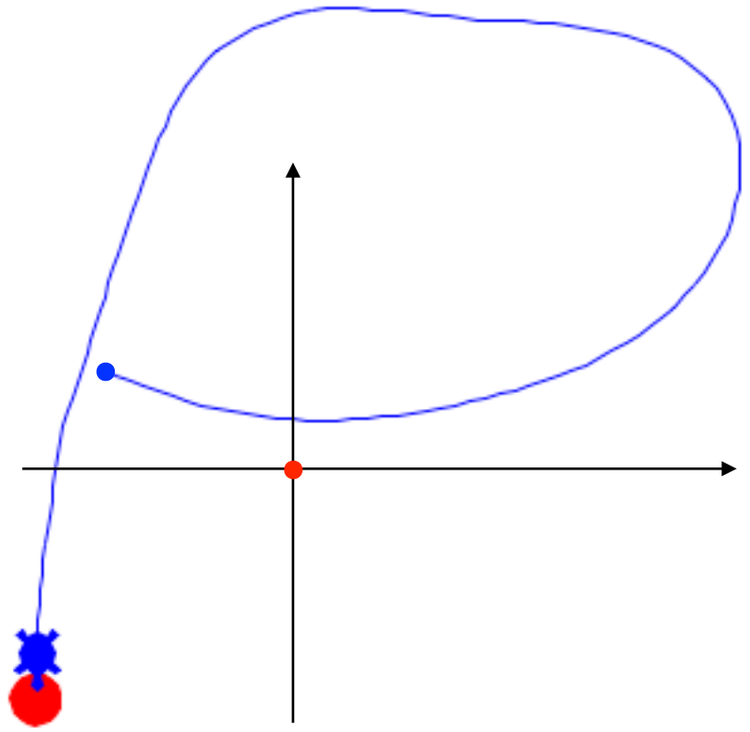
• Cette écriture est liée à un style : la **programmation par objets**, qui n'est pas au programme ! Les objets Python sont groupés en classes (str, …) et chaque classe contient des **fonctions** et des **méthodes** propres à la classe. Nous n'en dirons **pas plus**, nous bornant à une simple convention d'écriture, d'accord ? En général `fonction(x,y,…)` et plus rarement `objet.méthode(x,y,…)`.

• L'aide Python `>>> help(str.upper)` ou votre prof sont incontournables…

Des **objets** : les tortues ! *Optionnel !*

• Si l'on se contente d'**une** tortue (jusqu'à présent), on n'a pas besoin de la nommer. Mais si l'on veut simuler par exemple de problèmes de poursuites, il faut pouvoir **nommer chaque tortue**, qui sera un **objet** de la **classe** Turtle du **module** turtle.

```
# exemples_tortues.py
import turtle                # et non :  from turtle import *
from random import randint

lea = turtle.Turtle()        # Une nouvelle tortue
lea.color('red')             # rouge
lea.shape('circle')          # ayant une forme de disque
lea.up()                     # qui trace pas ton chemin.
bee = turtle.Turtle()        # Une autre tortue
bee.color('blue')            # bleue
bee.shape('turtle')          # avec la forme d'une tortue.

bee.up() ; bee.goto(-100,50) ; bee.down()
while distance(lea.pos(),bee.pos()) > 20 :
    lea.forward(6)           # Cours, Lea !
    lea.left(randint(-5,10))  # Tourne, Lea !
    bee.setheading(bee.towards(lea.pos()))
    bee.forward(6)           # Bee, avance vers Lea !
```



*Quelques **méthodes** de la classe Turtle !*

Le point de vue d'une IA (sur poe.com) <<<<<<<<<<<<<<<<<<<<<<<<<<

Unicode est un standard de codage de caractères qui vise à représenter tous les caractères utilisés dans les systèmes d'écriture du monde entier. Son développement a commencé dans les années 1980, avec le premier standard publié en 1991. Avant Unicode, divers systèmes de codage (comme ASCII) limitaient la représentation des caractères, rendant difficile l'échange de données entre langues et plateformes. Unicode a été conçu pour surmonter ces limitations en fournissant un code unique pour chaque caractère, facilitant ainsi la communication multilingue et l'édition de texte dans divers langages. Aujourd'hui, **Unicode** est devenu le standard dominant pour le traitement de texte dans les environnements numériques.

La notation Unicode est généralement écrite sous la forme U+XXXX, où XXXX représente un code **hexadécimal**. Voici les composants :

➥ ***Hexadécimal*** *= base 16, chiffres 0,1,2,...,9,A,B,C,D,E,F*

- **U+** : Indique qu'il s'agit d'un code Unicode.
- **XXXX** : Une séquence de chiffres et de lettres (0-9, A-F) qui représente le point de code du caractère en notation hexadécimale.

➥ *65 s'écrit 41 en hexa et* `0x41` *en Python.*

Exemples

1. **A** : Le caractère "A" est codé comme U+0041. En décimal, cela correspond à 65. En Python `'\u0041'`.
2. **é** : Le caractère "é" est codé comme U+00E9. En décimal 233. En Python `'\u00E9'` ou `chr(0xE9)`.
3. 汉 : Le caractère chinois "汉" est codé comme U+6C49. En décimal 27721. En Python `chr(27721)`.

Conversion

➥ $27721 = \underline{\mathbf{9}} + \underline{\mathbf{4}} \times 16 + \underline{\mathbf{C}} \times 16^2 + \underline{\mathbf{6}} \times 16^3$ *avec* $\mathbf{C} = 12$

Pour utiliser cette notation dans des langages de programmation comme Python, on peut utiliser la fonction chr() avec le code décimal, ou utiliser une chaîne de caractères spécifiant le code en Unicode avec la syntaxe \uXXXX pour les caractères de base, et \UXXXXXXXX pour les caractères ayant des codes plus élevés.

La fonction hex() en Python permet de convertir un nombre entier en sa représentation hexadécimale sous forme de chaîne de caractères. Elle renvoie le résultat avec le préfixe 0x, indiquant qu'il s'agit d'une valeur hexadécimale. Exemple : `hex(65) == '0x41'` et `0x41 == 65`. La fonction hex() est l'analogue de la fonction `bin()` pour la numérotation binaire.

POUR ÊTRE SÛR D'AVOIR COMPRIS…

- Allez sur www.poe.org, prenez un compte gratuit et demandez à une IA :

Avec Python, afficher en japonais "Je lis un bon manga"

- Modifiez la fonction cesar(s,k) pour garder **intact** tout caractère qui n'est **pas** une majuscule non accentuée : `cesar('LE CIEL',5) == 'QJ HNJQ'`

- Programmez la fonction decesar(s,k) prenant un message **crypté** s avec la clé k et renvoyant le texte **en clair** : `decesar('HNJQ',5) == 'CIEL'`.

- **Décodez** d'urgence le message crypté : `'QJ HNJQ STZX YTRGJ XZW QF YJYJ'`

- Cherchez dans la doc de la classe `str` quelle est la **méthode** qui teste si un caractère est une **lettre** (majuscule ou pas, accentuée ou pas).

- Cherchez sur le **Web** (ou avec une **IA**) comment trouver le chiffre numéro $k \geq 0$ d'un entier naturel n, la numérotation commençant à gauche.

- Programmez une fonction renverser(n) renvoyant l'entier dont les chiffres sont ceux de n mais **à l'envers** : `renverser(63520) == 2536`.

- Proposez une formule renvoyant le nombre de chiffres en **binaire** de la somme p+q où p et q sont deux entiers naturels.

La récurrence en programmation

• Les programmeurs ont des **boucles**, les matheux non ! Ces derniers se basent excluvivement sur la notion de **fonction**. Et il se trouve que leur principe de **récurrence** permet aussi de construire des algorithmes. Comment ?

• Le programme de maths de 1ère comporte les **suites définies par récurrence**. Elles peuvent se présenter sous deux formes similaires :

$$u_n = \begin{cases} 2 & \text{si } n = 0 \\ 3u_{n-1} + 1 & \text{sinon} \end{cases} \quad \text{ou} \quad \Longleftrightarrow \quad \left| \begin{array}{c} u_0 = 1 \\ u_{n+1} = 3u_n + 1 \end{array} \right. \qquad (n \in \mathbb{N})$$

mieux pour la prog *mieux pour les maths*

• La définition de gauche se traduit immédiatement en Python, pas celle de droite qui fait apparaître n+1. Gardons à l'esprit qu'**une suite est une fonction** sur $\mathbb{N}$.

```
def u(n) :          # la fonction n ↦ u_n
    if n == 0 :
        return 2
    return 3*u(n-1) + 1
```

```
>>> (u(0), u(5))
(2, 607)
```

$u_0 = 2, u_5 = 607$

• Python autorise la construction d'un tuple **par compréhension** :

```
>>> tuple(u(k) for k in range(10))
(2, 7, 22, 67, 202, 607, 1822, 5467, 16402, 49207)
```

• Dans une **définition par récurrence**, il y a toujours au moins un **cas de base** et au moins un **cas général**. Donc obligatoirement un **if** :

```
def u(n) :          # une fonction définie par récurrence
    if n == 0 :                    # le cas de base
        return 2
    return 3*u(n-1) + 1            # le cas général
```

$u_5 = 3u_4+1$
$= 3(3u_3 + 1) + 1$
$= 3(3(3u_2 + 1) + 1) + 1$
$= 3(3(3(3u_1 + 1) + 1) + 1) + 1$
$= 3(3(3(3(3u_0 + 1) + 1) + 1) + 1) + 1$
$= 3(3(3(3(3x2 + 1) + 1) + 1) + 1) + 1$
$= 3(3(3(3x7 + 1) + 1) + 1) + 1$
$= 3(3(3x22 + 1) + 1) + 1$
$= 3(3x67 + 1) + 1$
$= 3x202 + 1$
$= 607$

Expansion

Contraction

```
# <-- cas de base n=0
```

• La **récurrence** est mathématiquement très **pure**, mais il est difficile de l'exécuter à la main car elle demande de la **mémoire**. Il faut mettre des calculs en attente...

• *SAUF si elle se présente sous la forme d'une "boucle mathématique" (pp. 37-38).*

- EXEMPLE. La somme **1+2+3+…+n** pour n ≥ 0 entier.

```
def somme1(n) :                          # n ≥ 0 entier
    if n == 0 :                          # cas de base
        return 0
    return somme1(n-1) + n               # cas général
```

```
>>> somme1(10)
55
```

$$\sum_{k=1}^{10} k$$

(1 + 2 + 3 + … n-1) + n

- La somme des entiers de [a ; b] pour a ≤ b entiers.

```
def somme2(a,b) :                        # a ≤ b entiers
    if a > b :                           # cas de base
        return 0
    return a + somme2(a+1, b)            # cas général
```

```
>>> somme2(5,10)
45
```

$$\sum_{k=5}^{10} k$$

5 + (6 + 7 + 8 + 9 + 10)

```
def somme3(a,b) :                        # a ≤ b entiers
    if a > b :                           # cas de base
        return 0
    return somme3(a, b-1) + b            # cas général
```

```
>>> somme3(5,10)
45
```

$$\sum_{k=5}^{10} k$$

(5 + 6 + 7 + 8 + 9) + 10

- La **factorielle** n!

```
def fac(n) :                          #  n ≥ 0 entier
    if n == 0 :
        return 1
    return n * fac(n-1)               #  récurrence non terminale
```

p. 36

```
>>> fac(20)
2432902008176640000
```

```
def fac_iter(n) :      #  n ≥ 0 entier
    res = 1
    for k in range(2,n+1) :
        res = res * k
    return res
```

- Le **PGCD** par l'**algorithme d'Euclide**.

```
def pgcd(a,b) :                       #  a,b entiers naturels
    if b == 0 :
        return a
    return pgcd(b, a % b)             #  récurrence terminale
```

```
>>> pgcd(8,12)
4
```

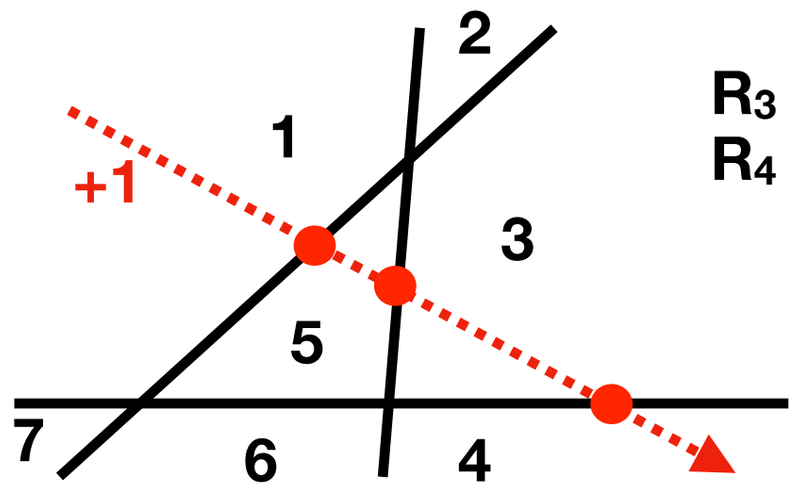
pgcd(8,12) == pgcd(12,8) == pgcd(8,4) == pgcd(4,0) == 4

```
def pgcd_iter(a,b) :       #  a,b entiers naturels
    while a > 0 :
        (a,b) = (b,a % b)
    return a
```

avec une boucle ⟺ ***itératif***

Raisonnement par récurrence (1)

• On considère n droites du plan *en position générale* (aucun couple de droites concourrantes ou parallèles). Nombre R_n de régions de plan ainsi délimitées ?



$R_3 = 7$
$R_4 = 11$

⇒ a) Si n = 0, alors $R_0 = 1$.

b) Supposons le résultat R_{n-1} connu pour n-1 droites, peut-on en déduire R_n ? Introduisons une **n-ème droite** dans la configuration. Elle arrive du lointain dans la région 1, en rajoutant déjà +1 nouvelle région. Ensuite, elle va couper chacune des n-1 droites en un seul point, et rajouter ainsi une nouvelle région pour chaque droite coupée. Donc $\boxed{R_n = R_{n-1} + 1 + (n-1) = R_{n-1} + n}$ STOP !

STOP ? Le mathématicien ne stoppe pas, il va essayer de résoudre la récurrence, trouver une formule pour le terme général de la suite (R_n). Le programmeur, lui, a **trouvé un algorithme** : $\boxed{R_0=1,\ R_n = R_{n-1} + n}$, qu'il améliorera peut-être par la suite…

```
def nb_regions(n) :
    if n == 0 :
        return 1
    return nb_regions(n-1) + n
```

$R_i = R_{i-1} + i$ *pour tout* $i=1..n$
Additionnons toutes ces équations :
$R_1 + \ldots + R_n = R_0 + \ldots + R_{n-1} + 1 + \ldots + n$
$R_n = R_0 + 1 + \ldots + n = 1 + n(n+1)/2$
$\boldsymbol{R_n = (n^2+n+2)/2}$

Raisonnement par récurrence (2)

- La **suite de Fibonacci** (F_n) est :

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34,…

Chaque terme F_n est la somme des deux précédents, sauf les deux premiers.

$$\begin{cases} F_0 & = 0 \\ F_1 & = 1 \\ F_n & = F_{n-1} + F_{n-2} \quad \text{si } n \geq 2 \end{cases}$$

- Aussitôt dit, aussitôt programmé :

```
def fib_rec(n) :      # par récurrence (double)
    if n < 2 :
        return n
    return fib_rec(n-1) + fib_rec(n-2)
```

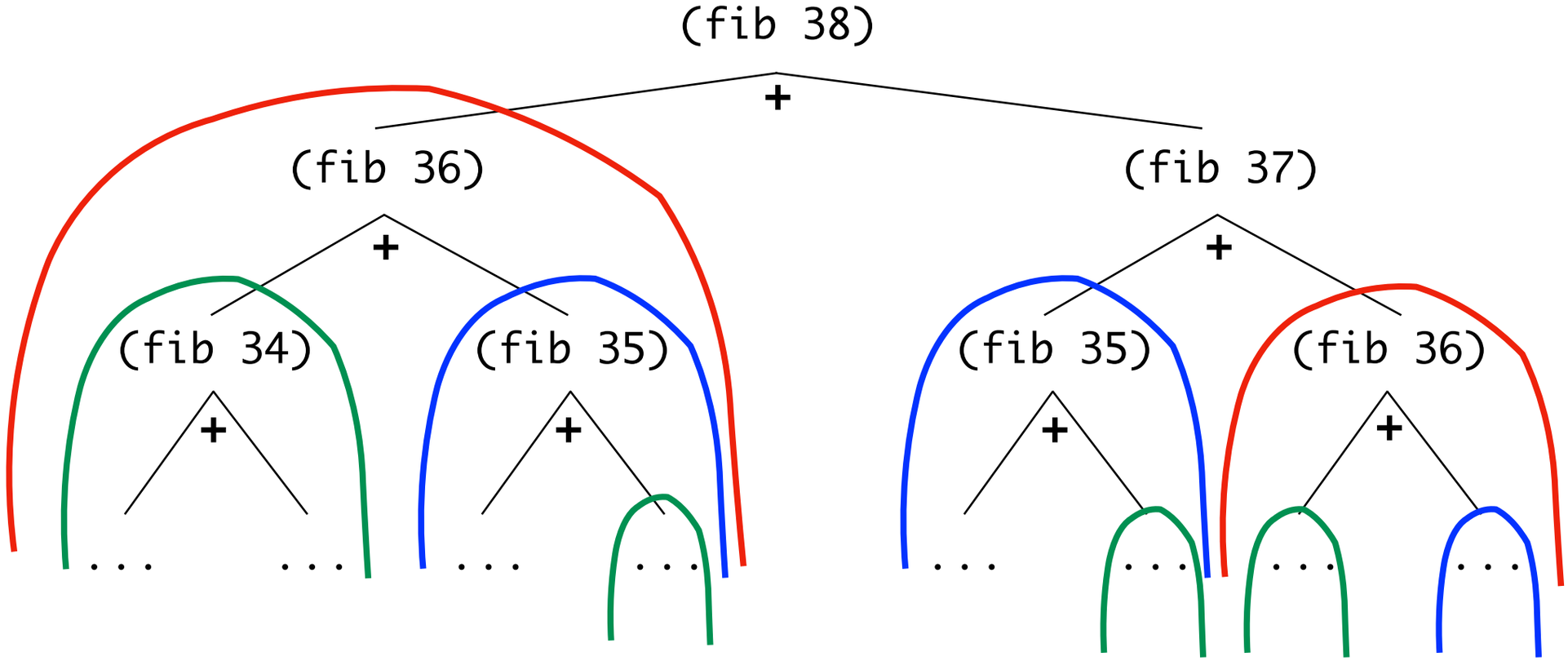
```
>>> tuple(fib_rec(k) for k in range(15))
(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377)
```

- Mais un **problème** surgit, qui différencie les maths et la programmation !

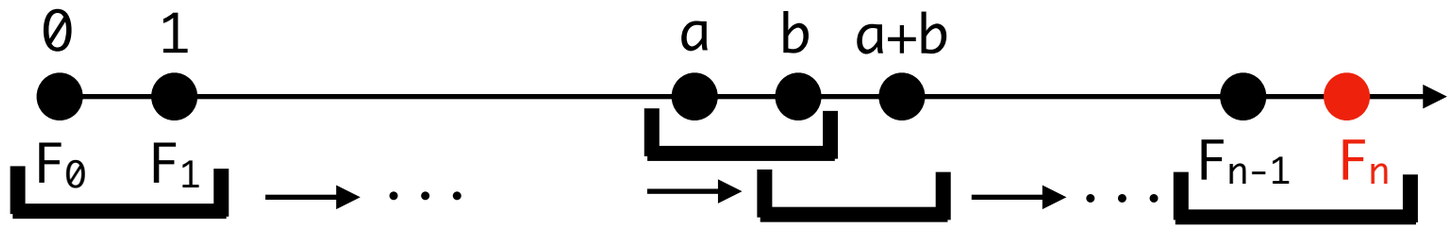
```
from time import time     # pour chronométrer

start = time()            # top chrono !
n = fib_rec(38)
end = time()              # top chrono !
print(f'{n} en {end-start} sec.')
```

```
39088169 en 9.39 sec.
```

• Regardons l'**arbre du calcul** : l'algorithme passe son temps à **faire et refaire les mêmes calculs** !! Mathématiquement élégant mais très inefficace…



• Tâchons d'obtenir une **boucle** calculant F_n en observant qu'il suffit de faire glisser n-1 fois un plateau contenant deux termes consécutifs F_i et F_{i+1} que nous nommons a et b.




```
39088169 en 1.9e-06 secondes
```

2 microsecondes !

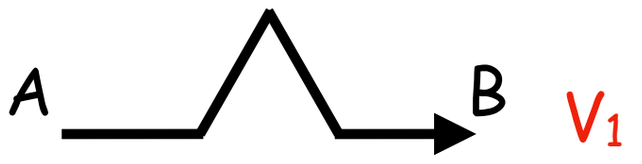
```
def fib_iter(n) :
    if n == 0 : return 0
    (a, b) = (0, 1)
    for k in range(n-1) :
        (a, b) = (b, a+b)
    return b
```

Raisonnement graphique par récurrence (3)

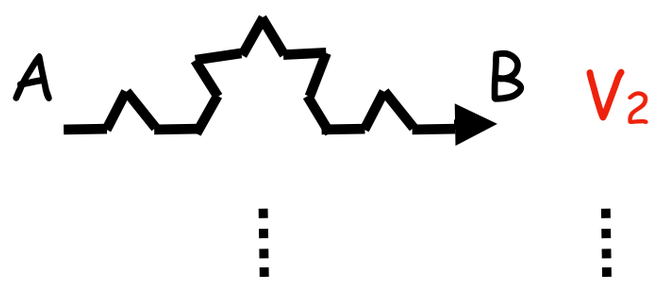
• La tortue se prête bien à des dessins par récurrence. Le plus célèbre est la **courbe fractale de von Koch** qui est la "limite" d'une suite de courbes (V_n).

• La courbe V_0 initiale est un segment AB. Longueur(V_0) = 1

• La courbe V_1 s'obtient en opérant sur V_0 la transformation suivante : on la divise en 3 segments égaux et l'on remplace le segment du milieu par un pic équilatéral. Longueur(V_1) = 4/3.



• **Récurrence** : La courbe V_n de niveau n s'obtient en opérant sur V_{n-1} la transformation V_0 —> V_1 sur chacun des sous-segments de V_n.



• Une fois saisi le mécanisme de génération, programmez **par récurrence** la fonction sans résultat vonKoch(n,x) qui **dessine** la courbe de niveau n ≥ 0, où x est la longueur du segment AB de départ. Testez avec n=3 et x = 200 par exemple.

• Modifiez-la pour qu'elle renvoie comme résultat la **longueur** de la courbe tracée. Cette longueur a-t-elle une limite ? Et l'aire entre la courbe et le segment AB ?...

Le point de vue d'une IA (sur poe.com) <<<<<<<<<<<<<<<<<<<<<<<<<<

Montrer le **codage récursif** à un élève de lycée présente plusieurs avantages :

1. **Liaison entre Mathématiques et Programmation**
   - La récursivité est un concept clé en mathématiques, notamment dans les suites récurrentes. Lier ces concepts aide les élèves à voir l'application pratique des mathématiques dans la programmation.
2. **Compréhension des Concepts Abstraits**
   - La récursivité aide à développer la pensée abstraite. Les élèves apprennent à décomposer des problèmes complexes en sous-problèmes plus simples, ce qui est essentiel en mathématiques et en informatique.
3. **Facilité d'Expression**
   - Pour certains problèmes (comme le calcul de la factorielle ou la suite de Fibonacci), la solution récursive peut être plus intuitive et plus simple à écrire qu'avec des boucles.
4. **Introduction à des Concepts Avancés**
   - La récursivité est souvent une porte d'entrée vers des concepts plus avancés, comme les algorithmes de recherche ou de tri (par exemple, tri rapide).
5. **Avantages de la Récursivité**
   - **Clarté et Simplicité** : Les solutions récursives peuvent être plus élégantes et plus faciles à comprendre.
   - **Utilisation Naturelle** : Certains problèmes, comme les arbres et les graphes, se prêtent mieux à une approche récursive.
6. **Inconvénients de la Récursivité**

surcharge de ressources

   - **Performance** : La récursivité peut être moins efficace en raison de l'*overhead* de l'appel de fonction et de la consommation de mémoire (pile d'appels).
   - **Limites de Profondeur** : Python a une limite de profondeur de récursion, ce qui peut poser problème avec des entrées très grandes.

POUR ÊTRE SÛR D'AVOIR COMPRIS…

- Allez sur www.poe.org, prenez un compte gratuit, demandez à une IA :
  Existe-t-il une formule pour calculer la somme des carrés jusqu'à n ?
  Et m'en fournir une preuve par récurrence.

- Pour calculer la dernière ligne $(u_0,u_1,u_2,\dots,u_9)$ de la p. 22, combien de **multiplications** ce calcul a-t-il fait ? Pouvez-vous calculer la somme $u_0 + u_1 + \dots + u_9$ avec une **dizaine** de multiplications seulement ?

- Programmer par **récurrence** une fonction `nbchif(n)` renvoyant le nombre de chiffres en base 10 de l'entier $n \geq 1$. Et en binaire ? Interdit d'utiliser la fonction `len` sur les chaînes…

- Programmez la fonction `chiffres(n,b)` de la p. 17 par **récurrence**.

- Programmez la fonction $n \longmapsto 1^3+2^3+3^3+\dots+n^3$ par **récurrence**, puis par boucle. Obtenir une formule mathématique polynomiale est un peu plus compliqué (IA ?)…

- Procédez au codage de la courbe de **von Koch** en p. 29. Si vous greffez cette courbe sur chaque côté d'un triangle équilatéral, vous obtiendrez le **flocon** de von Koch.

- Inventez un autre **dessin par récurrence**, joli si possible :-)

Algorithmes dans les entiers naturels

• Plus vous **lirez** et **coderez** des **algorithmes**, plus vous avancerez dans l'intuition du programmeur. L'*idéal* consiste à arriver au point où tout problème que l'on vous pose sera une déformation même lointaine d'un problème déjà traité. Gasp.

C'est un peu la démarche des IA génératives actuelles (GPT, Claude, Gemini, etc).

• Notre but : un bagage algorithmique minimum de Saison 2…

• **Définition** : Un entier n est **premier** si $n \geq 2$ et s'il est son seul diviseur > 1.

On examine à part le cas n=2 pour aller de 2 en 2 sur les impairs…

```
def ppdiv(n) :
    '''le plus petit diviseur d ≥ 2 de l'entier n ≥ 2'''
    if n % 2 == 0 : return 2
    d = 3
    while n % d > 0 :        #  d == n dans le pire des cas
        d = d + 2
    return d
```

```
>>> ppdiv(45)
3
>>> ppdiv(23)
23
```

NB. Notez que `ppdiv(n)` est toujours un nombre **premier**. Pourquoi ?…

Accélération (facile). On vérifie que s'il n'y a pas de diviseur de n dans [2,√n], il n'y en aura pas dans]√n ; n[donc le ppdiv sera n et n sera premier. On fera donc √n/2 divisions dans le pire des cas, celui où n est premier.

Accélération (bof). On peut faire mieux qu'aller de 2 en 2 en faisant un pas de 2 puis un pas de 4 puis un pas de 2 puis un pas de 4, etc. Pourquoi ? Parce qu'un nombre premier doit avoir un reste de division par 6 égal à ???. Vérifiez-le…

En bref :

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

Accélération (oups). Peut-on faire encore mieux ? Oui, mais il faut des maths de plus haut niveau, et oublier ppdiv… Wait and see.

• La **décomposition en facteurs premiers** !
Bizarrement, on ne dispose pas d'un algorithme général en 2024, fonctionnant sur les très grands nombres entiers. Nous nous contenterons d'une approche naïve, en calculant l'exposant d'un nombre premier p dans la décomposition de n, ce qui est très facile. Faites-le !

$$27744 = 2^5 \times 3 \times 17^2$$

```
def exposant(p,n) :
    res = ...
    while n % p ... :
        ...
    return res
```

```
>>> exposant(17, 27744)
2
```

• Pour la **décomposition**, notre tâche serait facilitée si l'on disposait d'une **liste** de nombres premiers. Nous allons donc parcourir les entiers à la recherche de chaque facteur premier pour en calculer l'exposant.

```
def affdecomp(n) :          #  affichage de la décomposition de n ≥ 2
    p = 2                   #  le premier candidat
    while n > 1 :
        e = exposant(p,n)   #  n = m . p^e
        if e > 0 :          #  donc p est un facteur premier
            n = n // p**e   #  n = m, donc n diminue…
            print(p,'**',e,sep='',end=' ')
        p = p + 1           #  prochain candidat ?
    print()
```

Pourquoi ?

```
>>> affdecomp(27744)
2**5 3**1 17**2
```

• Nous n'avons pas encore les listes, mais nous disposons des **tuples**. Sauriez-vous vous inspirer de affdecomp pour programmer decomp(n) qui n'affiche pas la décomposition mais la renvoie sous la forme d'un tuple résultat contenant la suite des facteurs premiers p et de leur exposant e ?

affichage ⇝ *stockage dans un tuple res*

print ⇝ *res = …*

```
>>> decomp(27744)
(2, 5, 3, 1, 17, 2)
```

à faire !

Intermède avec une suite arithmético-géométrique

• Reprenons la suite (u_n) de la page 21. Elle ressemble à la fois à une suite géométrique $u_n = 3u_{n-1}$ et à une suite arithmétique $u_n = u_{n-1} + 1$. Ces récurrences du 1er degré sont nommées **arithmético-géométriques**.

$$u_n = \begin{cases} 2 & \text{si } x = 0 \\ 3u_{n-1} + 1 & \text{sinon} \end{cases}$$

• L'addition +1 l'empêche d'être géométrique, enlevons-le par une translation en introduisant $v_n = u_n - \Delta$, et étudions la suite (v_n) :

$$v_0 = u_0 - \Delta = 2 - \Delta$$

$$v_n = u_n - \Delta = 3u_{n-1} + 1 - \Delta = 3(v_{n-1} + \Delta) + 1 - \Delta = 3v_{n-1} + \boxed{2\Delta + 1}$$

• Choisissons Δ tel que $2\Delta + 1 = 0$, soit $\Delta = -1/2$ de sorte que (v_n) devienne une suite **géométrique** de raison 3 et de premier terme 5/2, donc : $v_n = 5 \times 3^n/2$. A partir de là, on en déduit (u_n).

$$u_n = \frac{5}{2}3^n - \frac{1}{2}$$

• Ce "changement de variable" peut donner des idées sur la manière de manipuler un texte de programme. Certaines variations peuvent donner des résultats qualitatifs étonnants, comme la suivante…

Une transformation de récurrence. *Suite des pp. 21-22*

$$(*)\ \begin{cases} u_0 = 2 \\ u_n = 3u_{n-1} + 1 \end{cases}$$

• Intéressons-nous à la fonction $\boxed{v(n,a,b) = au(n) + b}$ pour généraliser la forme de la récurrence. Pouvons-nous trouver une définition pour v analogue à (*) ?

$v(0,a,b) = au(0) + b = 2a + b$

$v(n,a,b) = au(n) + b = a(3u_{n-1} + 1) + b = 3au_{n-1} + a + b = v(n-1, 3a, a+b)$

```
def v(n,a,b) :
    if n == 0 :
        return 2*a + b
    return v(n-1, 3*a, a + b)
```

et u(n) = (v(n,a,b) - b) / a

• D'où une nouvelle définition de u :

```
def new_u(n) :
    return (v(n,3,1) - 1) / 3
```

```
>>> tuple(new_u(k) for k in range(10))
(2.0, 7.0, 22.0, 67.0, 202.0, 607.0, 1822.0, 5467.0, 16402.0, 49207.0)
```

• Alors, je prétends que v simule une **boucle** au sens où :

```
v(5,3,1) = v(4,9,4)
         = v(3,27,13)
         = v(2,81,40)
         = v(1,243,121)
         = v(0,729,364)
         = 2*729 + 364
         = 1822
```

Pas d'expansion-contraction *comme en p. 22 ! Ce que vous voyez, c'est l'évolution des trois variables d'une boucle cachée !*

Récurrence **terminale**

u_5 = (v(5,3,1) - 1) / 3 = 607.0

– *Si c'est bien l'évolution des variables d'une boucle, on peut la rédiger en Python, cette boucle, capitaine ?*

– *Oui !*

récurrence

```
def v(n,a,b) :
    if n == 0 :
        return 2*a + b
    return v(n-1,3*a,a+b)
```

boucle (itération)

```
def new_v(n,a,b) :
    while n > 0 :
        (n,a,b) = (n-1,3*a,a+b)
    return 2*a + b
```

• Cette transformation *immédiate* entre récurrence et boucle n'est vraie **que** si l'appel de la fonction à elle-même n'est suivi d'aucun calcul (**récurrence *terminale***).

• En **Python**, la version avec boucle reste quand même **plus efficace**. Dans d'autres langages, la récurrence terminale aura la même efficacité qu'une boucle.

• En portant le texte de new_v dans u(n) = (v(n,a,b) - b) / 3,

il vient la nouvelle définition de u :

Magic !

```
def new_new_u(n) :
    a = 3 ; b = 1
    while n > 0 :
        (n,a,b) = (n-1,3*a,a + b)
    res = 2*a + b
    return (res - 1) // 3
```

```
>>> tuple(new_new_u(k) for k in range(10))
(2, 7, 22, 67, 202, 607, 1822, 5467, 16402, 49207)
```

MORALE. Nous avons effectué une **transformation de programme** sur un mode que l'on pourrait qualifier d'algébrique. A partir d'une suite définie par récurrence, nous avons rédigé une **fonction récursive** ($\Longleftrightarrow$ définie par récurrence). Avec un peu d'inventivité (généralisation de $3u_{n-1} + 1$ à $au_{n-1} + b$) et quelques manipulations algébriques, nous sommes parvenus à une version **récursive terminale** de la suite u, aussitôt transformée en boucle **while**. *[Mais c'est rarement aussi facile !...].*

• Le but était de transformer un texte de fonction. Et comprendre le cas particulier de la **récurrence terminale** qui donne en fait naissance à une sorte de ***boucle mathématique***. *Ceci dit, il était immédiat de transformer une récurrence de la forme $u_n = f(u_{n-1})$ en une boucle* for, *n'est-ce pas ?...*

voir bas de la page 40

Python et la récurrence (terminale ou pas ?)

• Mettons-nous bien d'accord. La **récurrence non terminale** a besoin de **mémoire**, car elle demande beaucoup de **calculs intermédiaires**. Prenons la factorielle (p. 24) :

fac(5) = 5*fac(4) = 5*4*fac(3) = 5*4*3*fac(2) = 5*4*3*2*fac(1) = 5*4*3*2*1*fac(0)

• Les calculs n'ont toujours pas commencé ! Nous somme sur le **cas de base** fac(0)=1.

fac(5) = 5*4*3*2*1*1

• Ces calculs intermédiaires **stockés en** mémoire vont maintenant être effectués :

fac(5) = 5*4*3*2*1 = 5*4*3*2 = 5*4*6 = 5*24 = 120 STOP !

• Cette zone mémoire de la machine se nomme une **PILE** (dernier entré, premier sorti). Si la pile monte trop haut (cela dépend de la machine), elle explose : BOUM.

```
>>> fac(3000)
RecursionError: maximum recursion depth exceeded
```

• Une boucle **while** n'a pas ce problème : son état n'est pas dans une pile mais dans une poignée de variables de boucle.

• Qu'en est-il de la **récurrence terminale** ? La théorie et certains langages de programmation disent qu'elle n'a **pas besoin** de pile, mais Python utilise quand même une pile qui ne sert à **rien** et donc… BOUM. Dont acte, mais ce n'est pas grave puisque la récurrence terminale se traduit facilement en boucle !

• L'euréka d'une fonction **récursive terminale** est de placer le **résultat** parmi les paramètres de la fonction. Reprenons l'exemple de $u_n = f(u_{n-1})$ en p. 22.

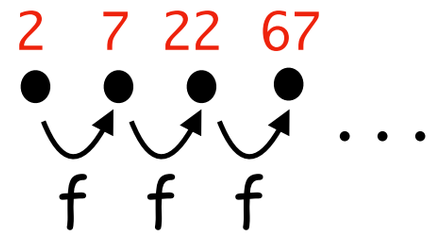
```
def u(n) :      # récursive non terminale
    if n == 0 :                 # cas de base
        return 2
    return f(u(n-1))
```

Résultat non nommé

n joue un simple rôle de compteur

$u_n = f(u_{n-1})$

$f(x) = 3x + 1$




```
def ut(n,res) :    # récursive terminale
    if n == 0 :                 # fini ?
        return res
    return ut(n-1,f(res))
```

Résultat en paramètre, initialisé à l'appel

```
ut(3,2) = ut(2,7)
        = ut(1,22)
        = ut(0,67)
        = 67
```

```
def uit(n) :                    # itérative
    res = 2
    while n > 0 :
        (n,res) = (n-1,f(res))
    return res
```

Résultat en variable locale

```
for k in range(n) :
    res = f(res)
```

• Méthodologie : pour penser une boucle, il est assez sain d'y voir l'**évolution du tuple des variables de boucle**. Prenons l'exemple de la factorielle fac(n) en page 24. Nous introduisons le résultat res en paramètre. Au fur et à mesure où on épuise la donnée n, le résultat res se remplit. *Vases communicants...*

```
def fac(n) :          # rec. non terminale
    if n == 0 :
        return 1
    return fac(n-1) * n
```

```
fac(4) = fac(3) * 4
       = fac(2) * 3 * 4
       = ...
       = 24
```

```
def fact(n,res) :          # rec. terminale
    if n == 0 :
        return res
    return fact(n-1,res*n)
```

```
fact(4,1) = fact(3,4)
          = fact(2,12)
          = fact(1,24)
          = 24
```

Evolution du tuple (n,res) des variables de boucle

```
def facit(n) :                    # itérative
    res = 1
    while n > 0 :
        (n,res) = (n-1,res*n)
    return res
```

```
(n,res) = (n-1,res*n)
```

ATTENTION !

```
n = n-1
res = res*n
```

Le point de vue d'une IA (sur poe.com) <<<<<<<<<<<<<<<<<<<<<<<<

La **récurrence terminale** est un concept en informatique et en mathématiques qui désigne une forme particulière de récursion où l'appel récursif est la dernière opération effectuée dans la fonction. Cela permet d'optimiser l'utilisation de la mémoire, car le compilateur ou l'interpréteur peut remplacer la pile d'appels récursifs par une simple boucle. Cela se traduit par une exécution plus efficace, notamment en évitant le dépassement de la pile (stack overflow).

1. Ressemblance avec une boucle en Python

En Python, une **boucle** (comme for ou while) et une fonction récursive terminale peuvent accomplir des tâches similaires. Exemple : le calcul de 1+2+…+n peut être réalisé avec une boucle, ou une fonction récursive terminale.

```python
def somme_boucle(n):
    total = 0
    for i in range(n + 1):
        total = total + i
    return total
```

```python
def somme_recursive_terminale(n, total=0):
    if n == 0:
        return total
    return somme_recursive_terminale(n - 1, total + n)
```

Dans cet exemple, la fonction somme_recursive_terminale fait un appel récursif en dernier, ce qui en fait une fonction à récurrence **terminale**.

"total = 0" signifie que l'argument "total", s'il est absent, sera mis à 0

2. Historique

L'idée de la récurrence terminale remonte aux travaux sur la récursion en informatique, qui ont été influencés par des concepts mathématiques et logiques. Elle est souvent associée à des langages de programmation comme Lisp et Scheme, qui ont promu des techniques de compilation et d'optimisation récursive dès les années 1970. Le concept a été formalisé et popularisé dans le cadre de la programmation fonctionnelle.

3. Conclusion

La **récurrence terminale** offre une approche efficace pour résoudre certains problèmes, **semblable à l'utilisation de boucles**, tout en tirant parti des avantages de la récursion. C'est un outil précieux dans l'arsenal d'un développeur, surtout dans le contexte de langages qui supportent cette optimisation.

POUR ÊTRE SÛR D'AVOIR COMPRIS…

- Prouvez que **le plus petit diviseur ≥ 2** d'un entier naturel n est premier (p. 32).
- Programmez la première **accélération de ppdiv** en haut de la page 33.
- Un entier naturel n est dit **parfait** s'il est égal à la somme de ses diviseurs : 6 est parfait (6 = 1+2+3) mais 8 ne l'est pas (8 ≠ 1+2+4). Programmez `est_parfait(n)` testant si n est parfait. Faites afficher les trois plus petits nombres parfaits.
- Programmez la fonction `decomp(n)` au bas de la p. 34.
- Programmez la fonction `comp(t)` telle que `comp(decomp(n)) == n`.

- Codez une version **récursive terminale** de la fonction `division` (page 9).

- **Arithmétique de Peano**. N'utilisons pas les opérateurs `+ - * / // % ==`

et ne conservons que `zero`, `add1` et `sub1` définies sur $\mathbb{N}$ par :

```
def zero(n) :
    return n == 0
```

```
def add1(n) :
    return n+1
```

```
def sub1(n) :   # n > 0
    return n-1
```

a) Programmez **par récurrence** `add(a,b)` renvoyant a+b (réc. terminale ou pas).

b) Quel est le nombre d'appels à add1 et sub1 lors du calcul de add(a,b) ?

c) Programmez **par récurrence** `mul(a,b)` renvoyant le produit ab.

d) Reprogrammez `add(a,b)` et `mul(a,b)` de manière **itérative** (boucle).

Le principe de **dichotomie**

• Le mot **dichotomie** désigne littéralement l'action de **couper en deux**. Il vient du grec ancien **dichotomia**, qui est composé de deux parties : **"dicha"** (δίχα) signifiant *en deux* ou *séparé*, et **"temnein"** (τέμνειν) signifiant *couper* (dixit mon IA…).

• En programmation, il s'agit d'un concept des plus importants, pour **construire** un algorithme, ou essayer d'**accélérer** un algorithme existant. Couper en deux, traiter chaque partie, rassembler les résultats. Une bonne stratégie à envisager !

• EXEMPLE 1. Calcul d'une **puissance** x^n, avec $n \geq 0$ entier.

Algorithme naïf, **linéaire** : $x^9 = x \times x \times x \times x \times x \times x \times x \times x \times x$

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \times x^{n-1} & \text{sinon} \end{cases}$$

```
def puis_rec(x,n) :
    if n == 0 :
        return 1
    return x * puis_rec(x,n-1)
```

```
def puis_iter(x,n) :
    res = 1
    for i in range(n) :
        res = res * x
    return res
```

```
>>> puis_iter(2,9)
512
>>> puis_rec(2,9)
512
```

2^{100} consomme 99 multiplications.

Algorithme **dichotomique** (1) : on coupe l'exposant en deux.

$$
\begin{array}{lrcll}
① & x^0 & = & 1 & \\
② & x^n & = & (x^2)^{n/2} & si\ n\ est\ pair \\
③ & x^n & = & x \times (x^2)^{(n-1)/2} & si\ n\ est\ impair
\end{array}
$$

$2^8 = (2^2)^4$
$2^9 = 2 \times (2^2)^4$

`4 == n // 2`

```
def puis_dicho(x, n) :
    if n == 0 :
①       return 1
    if n % 2 == 0 :
②       return puis_dicho(x*x, n//2)
③   return x * puis_dicho(x*x, n//2)

for n in (10,11,100) :
    print(f'2**{n} = {puis_dicho(2,n)}')
```

```
2**10 = 1024
2**11 = 2048
2**100 = 1267650600228229401496703205376
```

- Vérifiez (à la main ou mieux en codant) que

2^{100} utilise 10 multiplications seulement !!!

- **La DICHOTOMIE est l'une des stratégies les plus efficaces** : diviser pour régner !

...lorsque cette stratégie est possible !

Algorithme **dichotomique** (2). Et pourquoi pas une **boucle** ?

• La **dichotomie** est assez **rapide** pour ne pas avoir besoin de produire une **boucle** (souvent préférée à la récurrence en Python). Boucle plus **difficile** à coder en général…

• Mais on **VEUT** une boucle ! Comment la produire ? Soit en trouvant un tuple de **variables de boucle** adéquates, et après un certain nombre de mises au point car dans l'univers des boucles, c'est très facile ou très difficile suivant les problèmes. Soit en essayant de déformer une solution **récursive** déjà obtenue, car un grand pas a déjà été fait.

• Reprenons l'idée de la transformation des pages 32-34 et essayons de généraliser le problème. Dans les cas ② et ③, trouvons une forme commune aux deux résultats : ils calculent tous les deux une forme plus générale $a \times b^n$ (avec a=1 pour le cas ②).

Introduisons donc la fonction $f(n,a,b) = a\ b^n$ et cherchons une définition par récurrence de f à partir du système ① ② ③ :

n =0 —> $f(0, a, b) = a\ b^0 = a$

n pair —> $f(n, a, b) = a\ b^n = a\ (b^2)^{n//2} = f(n//2, a, b^2)$

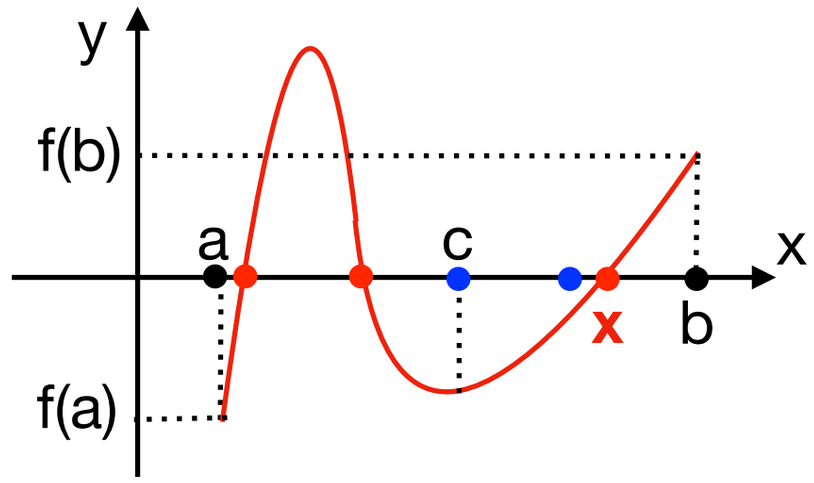
n impair —> $f(n, a, b) = a\ b^n = a\ (b\ (b^2)^{n//2}) = ab\ (b^2)^{n//2} = f(n//2, ab, b^2)$

• Nous reconnaissons là une forme de récurrence **terminale**, que l'on peut transformer à vue en une boucle **while n > 0** (pp. 39-40). Je vous laisse produire `puis_dicho_iter(x,n)` en Python. Nous n'avons quasiment pas gagné en efficacité ici, mais nous pouvons cette fois l'exécuter **pas à pas** à la main sans avoir besoin de mémoire à long terme…

Algorithme **dichotomique** (3). Dans le domaine **flottant**…

• Calcul d'une **solution** (**approchée**) de l'équation **f(x) = 0** où f est une fonction donnée. Il y a plusieurs approches possibles.

• Supposons que f soit **continue** sur [a,b], avec **f(a) * f(b) ≤ 0**. Alors le *théorème des valeurs intermédiaires* assure qu'il existe au moins un point $x \in [a,b]$ tel que f(x) = 0.



• La dichotomie divise [a,b] en son **milieu** c puis regarde le signe de f(a) * f(c).

- si f(a) * f(c) ≤ 0, alors on cherche une solution dans [a,c]
- sinon on cherche une solution dans [c,b].

• A force de diviser l'intervalle de recherche en deux, on va **encadrer une solution**. Il suffit de fixer la **précision** souhaitée h, largeur de l'encadrement final.

```
>>> from math import sin, pi
>>> sol_dicho(sin,2,11,0.01)
9.424769999999999
>>> 3 * pi
9.42477796076938
```

```
def sol_dicho(f,a,b,h) :
    while b - a > h :
        c = (a + b) / 2
        if f(a) * f(c) <= 0 : b = c
        else: a = c
    return (a + b) / 2
```

Ex : `sol_dicho(lambda x: x*x-2,0,5,0.0001)` ⇝ $\sqrt{2}$ *approchée (en 16 tours de boucle)*

Le point de vue d'une IA (sur poe.com) <<<<<<<<<<<<<<<<<<<<<<<<<

Aborder la **dichotomie** dans un cours de Python au lycée présente plusieurs intérêts pédagogiques :

1. **Concepts Fondamentaux de l'Algorithmique**
   - La dichotomie, ou recherche binaire, est un excellent moyen d'illustrer des concepts fondamentaux en algorithmique, tels que les algorithmes de recherche et de tri.
2. **Efficacité des Algorithmes**
   - En montrant comment la recherche binaire réduit le nombre de comparaisons nécessaires par rapport à une recherche linéaire, les élèves comprennent l'importance de l'efficacité algorithmique.
3. **Développement de l'Esprit Critique**
   - La dichotomie encourage les élèves à réfléchir de manière critique sur les méthodes de recherche et à évaluer les avantages et les inconvénients de différentes approches.
4. **Application Pratique**
   - Les élèves peuvent appliquer la recherche binaire dans divers contextes, comme la gestion de données ou le développement de jeux, rendant l'apprentissage plus concret.
5. **Introduction aux Structures de Données**
   - La dichotomie nécessite des données triées, ce qui peut être lié à l'enseignement des structures de données comme les tableaux et les listes.
6. **Programmation et Débogage**
   - Implémenter la recherche binaire en Python aide les élèves à développer des compétences en programmation et en débogage, essentielles pour leur avenir académique et professionnel.
7. **Préparation aux Examens et Compétitions**
   - La compréhension des algorithmes de recherche, y compris la dichotomie, est souvent un sujet d'examen et un thème récurrent dans les compétitions de programmation.

POUR ÊTRE SÛR D'AVOIR COMPRIS…

- Qu'est-ce que la **dichotomie** ?
- Pourquoi est-elle en général **efficace** en programmation ?
- Dans la **recherche par dichotomie** d'une solution de **f(x) = 0** sur [a,b], on rétrécit l'intervalle de moitié et on relance la recherche dans l'intervalle rétréci. Cela ne ressemblerait-il pas à une **récurrence** ? Coderiez-vous une recherche récursive ?…

page 46

- Trouvez <u>une</u> solution dans $\mathbb{R}$ à l'**équation** $2x^3 - x^2 + 2x - 1 = 0$. Pourquoi est-on certain qu'il y a au moins une solution ?
- Trouvez une solution approchée dans $[0,2\pi]$ à l'**équation** $\cos(x) = x$.
- Programmez `cout_de_puis_dicho(x,n)` renvoyant le **nombre de multiplications**.

page 44

- Supposez que j'ai un **tuple** `T` avec beaucoup de nombres dans le **désordre**. Puis-je procéder par dichotomie pour trouver si le nombre n=53 est dans ce tuple ?
- Supposez que j'ai un **tuple** `t` avec beaucoup de nombres en **ordre croissant**, donc `t[i]` ≤ `t[i+1]` pour tout i. En vous inspirant de la technique utilisée en page 46, programmez une fonction `appartient(n,t)` retournant True si et seulement si n est un élément de `t`. Technique : **essayez d'encadrer l'indice de n** dans `t`.

```
T = tuple(10*k for k in range(50))
```

Une nouvelle collection : la **LISTE**

• Une **liste** `L = [a,b,c]` ressemble à un tuple `(a,b,c)`. Même fonction `len` pour la longueur, même accès `L[i]` à l'élément numéro $i = 0..n\text{-}1$ si L est de longueur n. Même opérateur + pour concaténer les listes. Le constructeur **en extension** `[a,b,…]` **évalue** les expressions a,b,…

```
>>> L = [2, -8.5, 7, 1+2]
>>> type(L)
<class 'list'>
>>> len(L)
4                          # 4 éléments
>>> L
[2, -8.5, 7, 3]
```

```
>>> print(L[0],L[2],L[3],sep='-')
2-7-3
>>> L = L + [10, 20]
>>> L
[2, -8.5, 7, 3, 10, 20]
>>> L1 = [2, y, 3]
NameError: name 'y' is not defined
```

• Comme pour les tuples, on peut aussi construire une liste **en compréhension** :

```
>>> list(n*n for n in range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> list(k for k in range(0,50) if k % 6 == 0)
[0, 6, 12, 18, 24, 30, 36, 42, 48]
```

NB. Pour les listes, on peut aussi écrire `[n*n for n in range(10)]`.

• Différence majeure avec les tuples : on peut **modifier** chaque élément d'une liste.

```
>>> L = [3, 7, 4, 1, 9]
>>> L[1] = L[3]
>>> L[3] = L[3] + 1
>>> L
[3, 1, 4, 2, 9]
```

```
>>> T = (3, 7, 4, 1, 9)
>>> T[1] = T[3]
'tuple' object does not support item assignment
```

affectation

• Exemple : **construire** la **liste** des nombres **impairs** d'une liste d'entiers L.

```
def impairs(L) :
    res = []
    for i in range(len(L)) :
        if L[i] % 2 != 0 :
            res = res + [L[i]]
    return res
```

```
def impairs(L) :
    res = []
    for x in L :
        if x % 2 != 0 :
            res = res + [x]
    return res
```

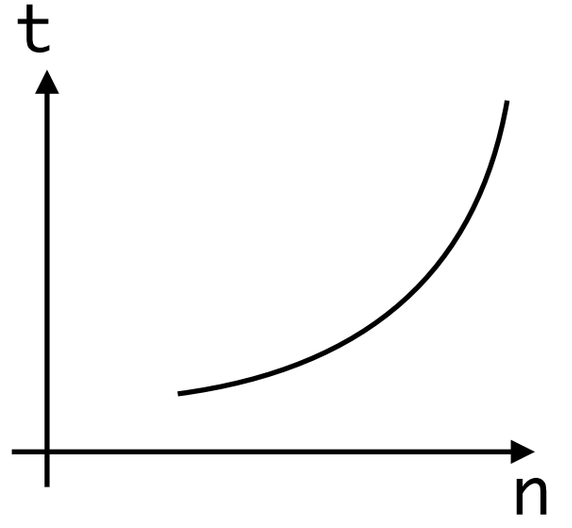
• La version de gauche est plus générale. Celle de droite n'est possible que si l'on n'utilise pas les numéros des éléments.

```
>>> impairs([3,7,6,5,8,0,4,9])
[3, 7, 5, 9]
```

Sensibilisation au coût d'un calcul

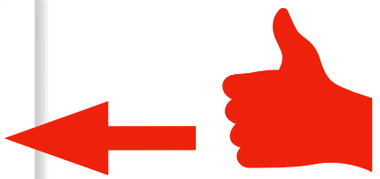
• Il faut savoir que la **concaténation** `L1 + L2` construit une **nouvelle** liste dont la longueur est `len(L1) + len(L2)`. Donc `res = res + [x]` coûte `len(res) + 1` en **temps** mais aussi en **espace** mémoire. Soit n la longueur de L.

• Dans le pire des cas (où tous les nombres de L sont impairs), le calcul `impairs(L)` coûtera donc 1+2+3+…+n = n(n+1)/2. On dit qu'il est de **coût quadratique** (polynôme du 2ème degré).
La croissance du coût est donc parabolique, trop rapide.
Ce serait mieux, *si c'est possible*, qu'il soit **linéaire** (1er degré).



• Or Python fournit un moyen d'**accrocher** un nouvel élément en queue (à droite) de la liste L avec un coût de 1 unité de calcul au lieu de n. Mais c'est une **méthode** `.append` de la classe **list**, sans résultat, et non une fonction usuelle (p. 18).

```
>>> L = [3, 7, 4, 1, 9]
>>> L.append(5)        # aucun résultat, coût = 1
>>> L
[3, 7, 4, 1, 9, 5]
```



CECI EST TRES IMPORTANT

```
>>> L = [3, 7, 4, 1, 9]
>>> L = L + [5]                    # coût = 6
>>> L
[3, 7, 4, 1, 9, 5]
```

• Nous pouvons donc accélérer la fonction impairs(L) :

```
def impairs(L) :
    res = []
    for x in L :
        if x % 2 != 0 :
            res.append(x)
    return res
```

```
>>> impairs([3,5,6,7,8,0,4,9])
[3, 5, 7, 9]
```

avec un coût linéaire !

ATTENTION : l'**accrochage physique en queue de liste** peut s'avérer **dangereux** si vous l'effectuez directement sur le paramètre L. Ici nous le faisons sur une variable **locale** res (une nouvelle liste), ce qui s'avère en général sain…

• **Filtrage** d'une liste L par un **prédicat**.

Un **prédicat** p est une fonction à résultat booléen (donc qui renvoie True ou False).

Filtrage : on ne retient que les éléments x de L vérifiant p, donc si p(x) est vrai.

```
def est_impair(n) :    # un prédicat
    return n % 2 == 1
```

```
def filtrage(L,p) :
    return [x for x in L if p(x)]
```

```
>>> filtrage([3,5,6,7,8,0,4,9], est_impair)
[3, 5, 7, 9]
```

*Quand vous pouvez **généraliser** une fonction, n'hésitez pas…*

- Ex : Calculer les nombres d'apparitions d'un élément dans une liste L.

```
def compter(x,L) :    # le nombre de fois que x apparaît dans L
    res = 0
    for y in L :
        if y == x :
            res = res + 1
    return res
```

Classique...

```
>>> liste = [4,7,3,2,3,3,1,19,3,-8]
>>> compter(3,liste)
4
```

```
def compter(x,L) :    # le nombre de fois que x apparaît dans L
    return len([y for y in L if y == x])
```

Ok

```
def compter(x,L) :    # le nombre de fois que x apparaît dans L
    return sum(1 for y in L if y == x)
```

Cool !

```
def compter(x,L) :    # le nombre de fois que x apparaît dans L
    return L.count(x)
```

*Ah mais oui, la méthode **count** pour les collections.*

Essayer d'abord de voir si l'on peut combiner des fonctions primitives du langage. Sinon, coder !...

La doc Python ? Un livre ? Une IA ?...

Le point de vue d'une IA (sur poe.com) <<<<<<<<<<<<<<<<<<<<<<<<<<

Les **listes** en Python sont des structures de données extrêmement utiles, surtout en mathématiques. Elles te permettent de stocker plusieurs valeurs sous une seule variable, ce qui simplifie la gestion des données. Par exemple, si tu souhaites garder une trace des notes d'un étudiant, tu peux créer une liste comme notes = [15, 12, 18, 10].

Chaque élément d'une liste a un index, ce qui te permet d'accéder facilement à des valeurs précises. Tu peux par exemple obtenir la première note avec notes[0]. De plus, les listes sont **mutables**, ce qui signifie que tu peux modifier leur contenu après leur création. Si tu souhaites ajouter une nouvelle note, tu peux utiliser notes.append(14) pour l'ajouter à la fin de la liste.

Les listes facilitent également l'itération grâce aux boucles. Cela est très utile pour effectuer des calculs sur chaque élément, comme la somme des notes. Par exemple, tu peux parcourir les notes et calculer la somme avec une simple boucle. Python propose aussi des fonctions intégrées comme **sum()** et **len()** qui rendent ces opérations encore plus simples.

En mathématiques, les listes peuvent représenter des structures comme des **vecteurs** ou des **matrices**. Par exemple, une matrice 2x2 peut être définie par une liste de listes, comme matrice = [[1, 2], [3, 4]]. Cela te permet de réaliser des calculs matriciels facilement.

Avec des bibliothèques comme **NumPy**, tu peux même travailler avec des listes de manière encore plus efficace. Ces bibliothèques offrent des fonctionnalités avancées pour effectuer des opérations mathématiques sur des tableaux.

Les listes sont également très flexibles, car elles peuvent contenir **différents types de données**, ce qui te permet de créer des structures variées pour tes calculs. Par exemple, tu peux mélanger des nombres et des chaînes dans une même liste si besoin.

Enfin, les listes sont idéales pour résoudre des problèmes mathématiques, comme stocker des séquences ou les termes d'une suite, comme la suite de Fibonacci. En résumé, utiliser des listes en Python te permet de gérer et manipuler des données mathématiques de manière efficace et organisée, rendant tes programmes plus puissants et flexibles.

POUR ÊTRE SÛR D'AVOIR COMPRIS…

• Qu'est-ce qui différencie un **tuple** et une **liste** ? Peut-on mettre **n'importe quel type** d'éléments dans une liste comme dans `[5, True, 8, ('x', 9), [3,4]]` ?

• Construisez la liste P1000 des **nombres premiers** jusqu'à 1000 (cf. page 29).

• Si L1, L2 et L3 sont des **listes**, on a : `L1 + (L2 + L3) == (L1 + L2) + L3`. Mais à quelle condition la construction de gauche est-elle **plus efficace** que celle de droite ? Raisonnez en termes de coûts (page 51).

• Programmez la fonction `sans_doublons(L)` retournant une copie de L dans laquelle chaque élément n'apparaît plus qu'une seule fois. Utilisez l'opérateur `in`.

• Programmez une fonction `tableau_pts(f,a,b,h)` renvoyant une **liste de points** `(x,y)` sur la courbe de `f`, pour `x = a, a+h, a+2h, …` dans l'intervalle `[a,b]`.

• Programmez une fonction `est_croissante(L)` retournant `True` si la liste L est ordonnée en croissant (au sens large), ou bien `False` sinon.

• Programmez une fonction `build_list(n,f)` retournant une liste de longueur n dont l'élément numéro k est `f(k)`. Deux solutions possibles…

• Programmez une fonction `separer(L)` prenant deux listes d'entiers et retournant un couple de deux listes `(L1, L2)` où `L1` contient les entiers pairs et `L2` les impairs, dans l'ordre où ils apparaissent dans `L`. Combien y a-t-il d'impairs ?

Quelques mots sur la résolution de problèmes

• C'est l'une des activités majeures du programmeur et du scientifique en général.

Il y a plusieurs approches possibles.

Pour obtenir un résultat exact :

- ✓ (1) Trouver une formule exacte : algorithme direct (rare).
- ✓ (2) Utiliser le principe diviser pour régner afin de se ramener à des sous-problèmes plus simples (dichotomie, récurrence).
- ✓ (3) Cerner l'état du système, le représenter par des variables et boucler sur ces variables jusqu'à l'obtention du résultat.

Pour se contenter d'un résultat approché :

- (4) Trouver une formule approchée : algorithme direct (rare).
- ✓ (5) Procéder par améliorations successives vers une solution assez bonne.
- (6) Faire appel à des moteurs d'Intelligence Artificielle (IA) basés sur la programmation génétique ou les réseaux de neurones (*deep learning*).

• Nous avons déjà abordé les points 1-2-3-5. Nous allons voir un exemple du point 4 puis approfondir le point 5. Le point 6 sera abordé dans le Supérieur.

Une formule approchée pour la dérivation numérique

• En maths la valeur de la **fonction dérivée de f au point a** est définie comme une limite de la **pente** de la corde joignant le point fixe A(a,f(a)) au point mobile M(a+h,f(a+h)), limite lorsque h s'approche de 0.

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

• En programmation (numérique), la limite n'existe pas, on se contente d'une valeur approchée pour **h voisin de 0**, par exemple 0.01. On garde la définition précédente en enlevant la limite !

$$f'(a) \approx \frac{f(a+h) - f(a)}{h} \qquad \textit{avec h petit et fixé}$$

✓ 4

```
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
```

si la fonction f est utile plusieurs fois :

```
def f(x) :
    return x**3 - 5*x**2 + 4
```

$$f(x) = x^3 - 5x^2 + 4$$

• Mais deriv n'est pas propice à calculer f"(1). On ne peut pas exprimer f" = (f')', il faudrait une formule approchée pour la dérivée seconde (elle existe !).

```
>>> f = lambda x: x**3 - 5*x**2 + 4
>>> df1 = deriv(f, 1, 0.001)
>>> deriv(df1,1,0.001)
TypeError: 'float' object is not callable
```

On essaye de dériver un nombre !

• Idée : on *fait abstraction* du paramètre x dans deriv pour rendre une **fonction** et non un nombre. Nommons D l'opérateur de dérivation approchée $f \mapsto f'$.

```
def D(f,h):                                  # approximation de f'
    return lambda x: (f(x+h) - f(x)) / h
```

```
>>> Df = D(f,0.001)          # ≃ f'
>>> D2f = D(Df,0.001)        # ≃ f''=(f')'
>>> D2f(1)                   # ≃ f''(1)
-3.9940000000626696          # exact : -4
```

NB. Le module simpy (SYMbolic PYthon) permet de calculer des dérivées exactes, mais il est totalement hors-programme au lycée en 2024. Voir le fichier lycee_saison_2.py

Une technique de Newton : les approximations successives

• Popularisée vers 1690, elle permet de **résoudre une équation** $f(x) = 0$, sous certaines conditions. Elle utilise la dérivation approchée pour former une suite (x_n) qui converge vers une abscisse r telle que $f(r) \simeq 0$.

• Idée générale : approcher une solution par des **approximations successives**.

Tentative de calcul d'une solution approchée :

Soit a une approximation de la solution.

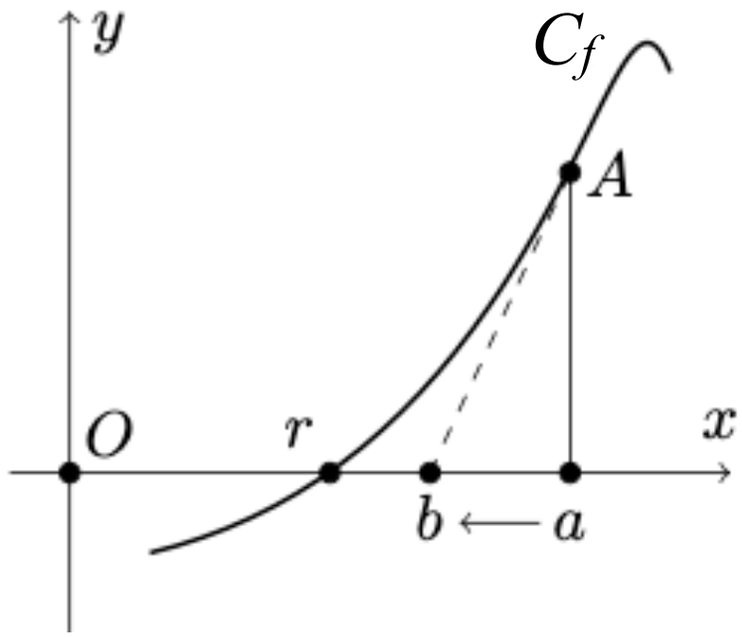
Tant que l'approximation a n'est pas assez bonne :

a ⟵ amélioration(a)

• *Avant d'appliquer la technique, il est sain d'avoir la courbe C_f sous les yeux pour apprécier ses variations.*

• **Assez bonne** ? On ne peut pas exprimer que a est proche de r puisqu'on ne connaît pas r. On exprimera que f(a) est presque nul.

• **Amélioration** ! On considère le point A, on trace la tangente à (C_f) en A qui coupe Ox en une abscisse b qui est une amélioration de a.



et si elle ne la coupe pas ?...

Le point de vue d'une IA (sur poe.com) <<<<<<<<<<<<<<<<<<<<<<<<<<

La méthode de Newton, aussi connue sous le nom de méthode de Newton-Raphson, est une technique itérative pour trouver les racines d'une fonction. Voici un aperçu de son histoire et développement :

Origines

Isaac Newton (1643-1727) : Bien que Newton soit souvent associé à cette méthode, il n'a pas directement formulé la méthode de Newton-Raphson sous cette forme. Ses travaux en calcul infinitésimal ont jeté les bases pour comprendre les fonctions et leurs dérivées.

Joseph Raphson (1668-1715) : Raphson a popularisé la méthode dans son ouvrage *Analysis Aequationum*, publié en 1690. Il a présenté la méthode sous une forme permettant de trouver les racines d'une équation polynomiale.

Développement et Applications

18ème siècle : La méthode a été largement adoptée par les mathématiciens et a été améliorée au fil du temps. Elle a trouvé des applications dans divers domaines, de la physique à l'économie.

19ème siècle : Des améliorations ont été apportées pour augmenter la convergence et la robustesse de la méthode. Des variantes ont été développées pour des fonctions plus complexes.

Principe de la Méthode

La méthode de Newton-Raphson repose sur l'idée suivante :

1. On commence avec une estimation initiale x_0 de la racine d'une fonction $f(x)$.
2. On calcule la prochaine approximation x_{n+1} en utilisant la formule : $x_{n+1} = x_n - .....$
3. On itère ce processus jusqu'à ce que la valeur converge vers la racine recherchée.

Exo !

Avantages : Convergence rapide pour des estimations proches de la racine. Pratique pour des fonctions dérivables.

Limitations : Peut ne pas converger si l'estimation initiale est trop éloignée de la racine. Sensible aux points critiques et aux dérivées nulles.

Le point de vue d'une IA (sur poe.com) <<<<<<<<<<<<<<<<<<<<<<<<<<

La **résolution de problèmes** remonte à des pratiques anciennes dans les domaines des mathématiques. Les mathématiciens de l'Antiquité, comme Euclide, ont établi des méthodes pour résoudre des problèmes géométriques, établissant ainsi les bases de la logique et de la pensée analytique.

Au cours de la période classique, qui s'étend du XVIIe au XIXe siècle, la résolution de problèmes a été formalisée. Des figures comme René Descartes et Gottfried Wilhelm Leibniz ont contribué à la compréhension systématique des problèmes, en introduisant des méthodes analytiques et algébriques.

Principaux développements :

- **Méthode cartésienne :** Une approche systématique pour décomposer les problèmes complexes en éléments plus simples.
- **Logique formelle :** Le développement de la logique a permis de formaliser les raisonnements, rendant résolution de problèmes plus rigoureuse.

Intelligence Artificielle

Avec l'émergence de l'IA dans la seconde moitié du XXe siècle, la résolution de problème a pris une nouvelle dimension. Des chercheurs comme Alan Turing et John McCarthy ont exploré comment les machines pouvaient simuler le raisonnement humain.

Avancées clés :

- **Algorithmes de recherche :** Développement d'algorithmes pour explorer des espaces de solutions (comme la recherche en profondeur et la recherche en largeur).
- **Résolution de problèmes par contraintes :** Méthodes permettant de résoudre des problèmes en respectant des contraintes spécifiques.
- **Apprentissage automatique :** Utilisation de données pour améliorer les processus de résolution de problème, permettant aux machines d'apprendre de nouvelles stratégies.

POUR ÊTRE SÛR D'AVOIR COMPRIS…

• La formule de la limite en page 58 est issue de la pente de la droite passant par les points M(x,f(x)) et N(x+h,f(x+h)). Une autre formule consiste à se placer de part et d'autre du point M, aux points d'abscisse x-h et x+h. Quelle est la formule obtenue et que deviennent les fonctions `deriv` et `D` ? Sont-elles plus précises ?

• Soit f une fonction dérivable. Programmez une fonction `tangente(f,a,h)` renvoyant un couple (a,b) tel que l'équation approchée de la tangente à la courbe de f au point d'abscisse a soit y = ax+b.

• La technique de Newton. Programmez la fonction `solve(f,a,h)` renvoyant une solution (on suppose qu'elle existe) de l'équation f(x) = 0 où f est une fonction dérivable, a est une approximation initiale de la solution et h est voisin de 0. Trouvez une solution aux équations $2x^3 - x^2 + 2x - 1 = 0$ et $x = \cos(x)$, cf. page 49.

• Dans le code de votre fonction `solve`, voyez-vous un endroit où l'on peut détecter une faille potentielle de la technique de Newton ? Comment l'interprétez-vous géométriquement ?

• Modifiez votre fonction `solve` pour qu'elle renvoie une **liste** de toutes les approximations `a` rencontrées avant l'obtention d'une solution qui sera le dernier élément de cette liste. Cela donnera aussi le nombre de tours de boucle.

Tracé d'une courbe avec **numpy** (numerical python)

• On peut tracer une courbe avec la tortue, mais le module **numpy** fournit une approche plus systématique. Nous considérons qu'une **courbe** est un ensemble de points (x,y) du plan, avec par ex. y = f(x) mais pas forcément (ex: un cercle).

LES TABLEAUX DE NUMPY

```
import numpy as np
```

Soit np *le module* numpy, *pour faire plus court !*

• La liste Python usuelle est remplacée en numpy par un **tableau** (*array*), qui ne contiendra que des **éléments de même type** (en général des nombres flottants), ce qui permet d'accélérer les calculs. np.xxx dénotera un objet (en général une fonction) qui provient de np. Par exemple les fonctions np.array, np.linspace, ...

• On peut construire un **tableau** à partir d'une **liste** :

```
>>> T = np.array([4,8,6,1])
>>> T          # T est un tableau
array([4, 8, 6, 1])
```

```
>>> len(T)
4
>>> T[2]
6
```

```
>>> T + 2*T                # Cool !
array([12, 24, 18,  3, 27])
>>> print(T + 2*T)
[12 24 18  3 27]           # Ok...
```

https://pyspc.readthedocs.io/fr/latest/05-bases/08-tableaux_numpy.html

• La fonction np.arange(a,b,s) pour les tableaux prend les mêmes arguments que range, mais retourne un tableau.

```
>>> print(np.arange(2,8))
[2 3 4 5 6 7]
```

```
>>> print(np.arange(2,8,2))
[2 4 6]
```

• La fonction np.linspace(a,b,n) retourne un tableau de n nombres espacés régulièrement dans [a ; b].

```
>>> np.linspace(0,2,8)                  # découpage de [0,2] en 8 points
array([0.  , 0.28571429, 0.57142857, 0.85714286, 1.14285714,
       1.42857143, 1.71428571, 2. ])
```

• Prenons une courbe, par exemple d'équation $y = x^3-2x+1$. Nous souhaitons la dessiner sur l'intervalle [-1 ; 2]. Nous commençons par fabriquer un tableau de points (disons 20 points) : un tableau d'abscisses Tx et un tableau d'ordonnées Ty.

```
>>> Tx = np.linspace(-1,2,20)
>>> Tx
array([-1. , -0.84210526 ,..., 2. ]
```

```
>>> Ty = Tx**3 - 2*Tx + 1
>>> Ty
array([ 2. , 2.08703893 ,..., 5. ]
```

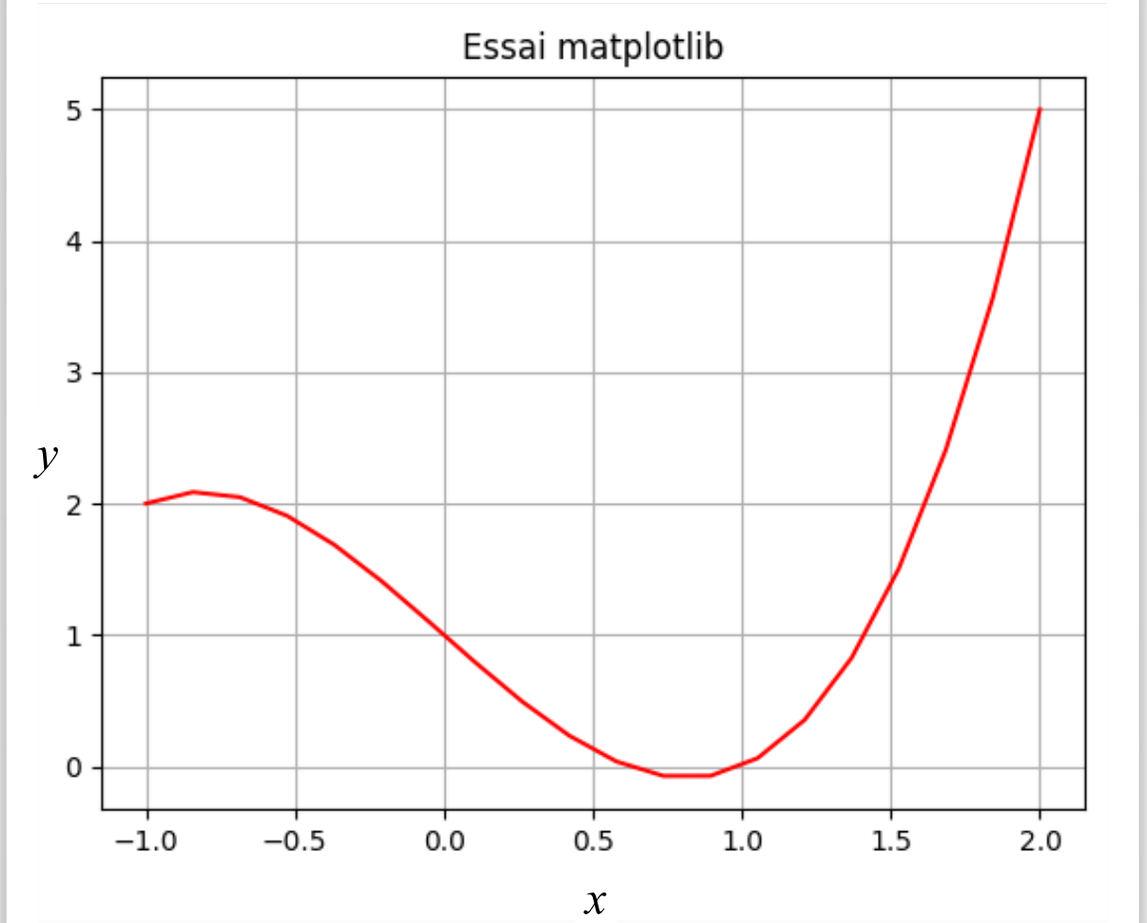
2. ⟺ 2.0 en Python

• Il suffit d'utiliser le module **matplotlib.pyplot** spécialisé dans les dessins mathématiques, et lui passer les tableaux Tx et Ty, ainsi que des informations décoratives optionnelles (couleurs, etc).

```
import matplotlib.pyplot as plt
```

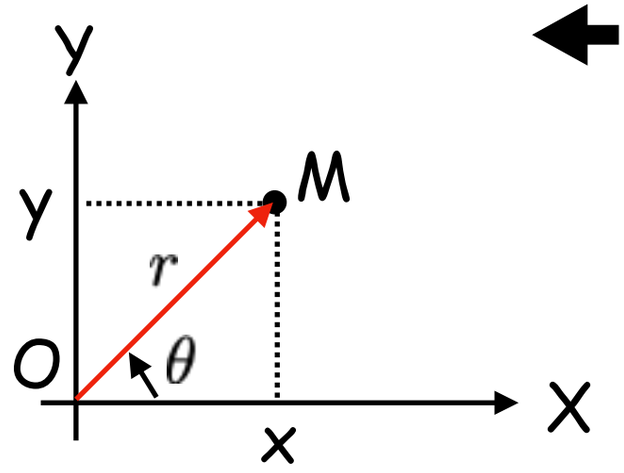
• Le programme complet pour tracer la courbe :

```
import numpy as np
import matplotlib.pyplot as plt

Tx = np.linspace(-1,2,20)
Ty = Tx**3 - 2*Tx + 1
plt.plot(Tx,Ty,color='r')  # red
plt.xlabel('x')
plt.ylabel('y')
plt.grid()                 # le grillage
plt.title('Essai matplotlib')
plt.show()                 # Show it now !
```



https://pyspc.readthedocs.io/fr/latest/05-bases/10-graphiques_partie_1.html

• Un point M du plan peut être repéré par ses coordonnées cartésiennes $M(x, y)$, ou bien par ses coordonnées polaires $\mathbf{M}(r, \theta)$ où r est la distance à l'origine et θ l'angle polaire de M.

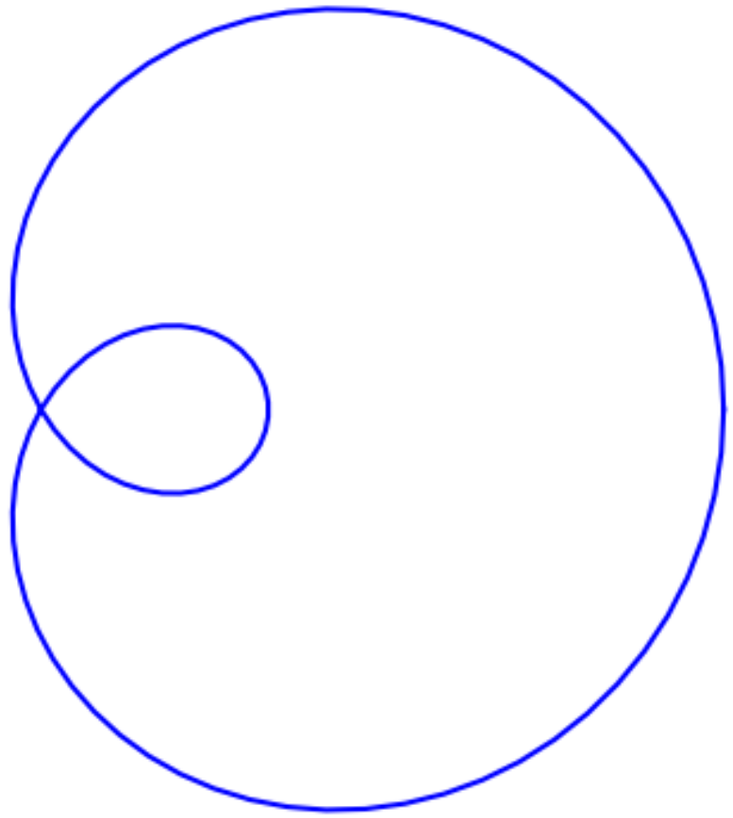


$$r = ||\vec{OM}|| \quad \text{et} \quad \theta = (\vec{Ox}, \vec{OM})$$

• Une **courbe polaire** est définie par une équation de la forme $r = f(\theta)$.
Prenons l'exemple de la courbe d'équation $r = 1 + 2\cos(\theta)$.

• Avec Matplotlib, on peut dessiner cette courbe polaire, pour un angle θ variant dans [0,2π], **en se ramenant aux coordonnées cartésiennes** par les formules bien connues des lycéens : $x = r\cos(\theta)$ et $y = r\sin(\theta)$.

```
def test_matplotlib_polaire() :
    theta = np.linspace(0, 2*np.pi, 100)
    r = 1 + 2*np.cos(theta)
    Tx = r*np.cos(theta)
    Ty = r*np.sin(theta)
    plt.axis("equal")     # repère orthonormé
    plt.plot(Tx,Ty,color='b')
    plt.show()
```



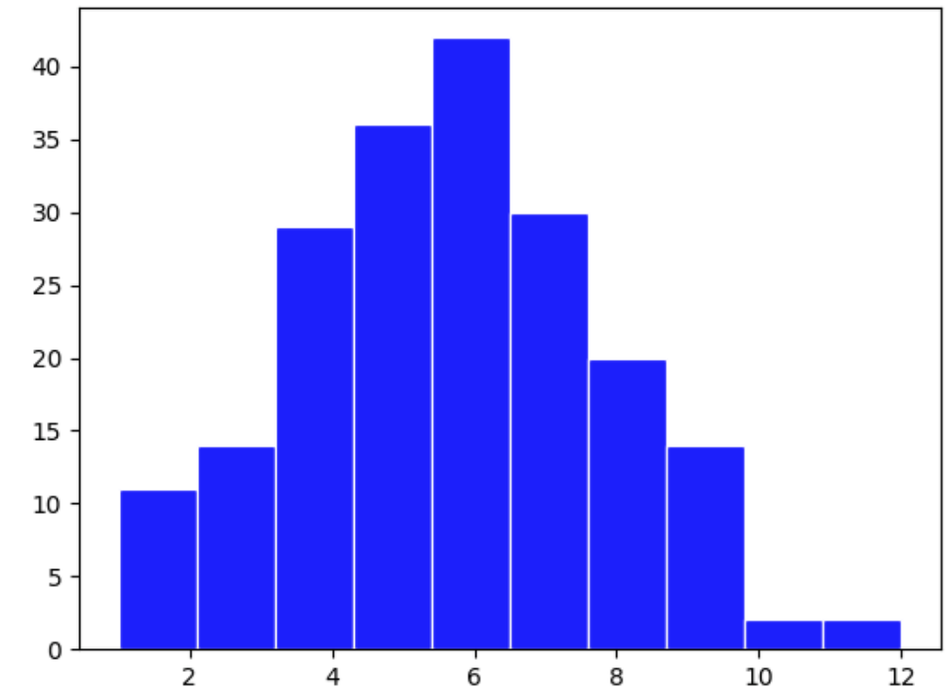
NB. On pouvait travailler directement en polaire avec plt.polar…

Matplotlib en **probas/stats**

• Lançons n=20 fois une pièce mal équilibrée et comptons le nombre de succès (p=0.3). Procédons à 200 telles expériences et visualisons l'**histogramme** de cette **série binomiale** avec 10 intervalles (*bins*).

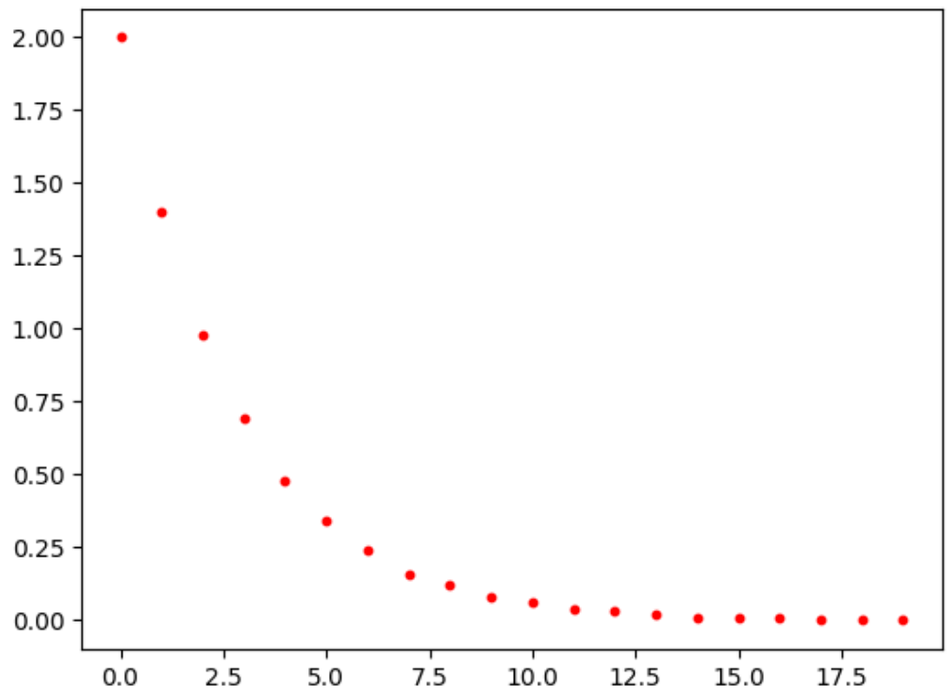
```
s = np.random.binomial(20,0.3,200)
a = plt.hist(s,bins=10,color='b',edgecolor='w')
plt.show()
```

```
>>> s
[ 6  4  6  11  2  8  6  ... 7  5   6  4  8  6]
>>> a
a = (array([11., 14., 29., ..., 14., 2.,  2.]),
     array([ 1. ,  2.1,  3.2, ..., 12. ])
```



• Construisons une **suite géométrique** de raison 0.7 et visualisons le **nuage de points**.

```
Ln = list(range(20))          # [0,1,2,...,19]
Un = [round(2 * 0.7**k, 2) for k in range(20)]
# print([(Ln[k],Un[k]) for k in range(20)])
plt.scatter(Ln,Un,s=10,color='r')
# plt.plot(Ln,Un,'ro')        # une autre manière
plt.show()
```



NB. On peut utiliser dans des cas très simples des listes au lieu de tableaux avec matplotlib…

Le point de vue d'une IA (sur poe.com) <<<<<<<<<<<<<<<<<<<<<<<<<<

Aborder Matplotlib avec un élève de lycée qui apprend Python présente plusieurs intérêts :

1. **Visualisation des données**: Matplotlib permet de créer des graphiques et des visualisations de données, ce qui est un outil puissant pour comprendre et analyser des informations. Cela permet aux élèves de visualiser les relations entre les données, de repérer des tendances et de tirer des conclusions.
2. **Renforcement de la compréhension des concepts mathématiques**: Matplotlib peut être utilisé pour représenter des fonctions mathématiques, des équations et des concepts géométriques. Cela permet aux élèves de visualiser ces concepts abstraits et de les comprendre plus facilement.
3. **Développement de compétences en programmation**: L'utilisation de Matplotlib implique l'écriture de code Python, ce qui renforce les compétences en programmation des élèves. Ils apprennent à utiliser des fonctions, des arguments et des structures de données pour créer des graphiques et des visualisations.
4. **Introduction à la science des données**: Matplotlib est un outil essentiel dans le domaine de la science des données. En utilisant Matplotlib, les élèves peuvent acquérir une base solide pour explorer des données et réaliser des analyses statistiques.
5. **Motivation et engagement**: La possibilité de créer des graphiques et des visualisations visuellement attrayantes peut motiver les élèves et les engager dans l'apprentissage de Python. Ils peuvent utiliser Matplotlib pour créer des projets créatifs et partager leurs résultats avec les autres.

En résumé, Matplotlib offre aux élèves de lycée une opportunité d'apprendre à visualiser des données, de renforcer leur compréhension des concepts mathématiques, de développer leurs compétences en programmation et de s'initier à la science des données. C'est un outil précieux pour l'apprentissage de Python et pour développer des compétences essentielles dans le monde numérique actuel.

POUR ÊTRE SÛR D'AVOIR COMPRIS…

• Programmez la fonction `polyreg_pol(n,c)` ne dessinant PAS un **polygone régulier** de n côtés de longueur c à partir de l'état courant (position, cap) de la tortue. Graphisme polaire. La fonction retournera la liste des sommets du polygone.

• Fonction `polyreg_cart(S)` : dessinez le polygone régulier précédent de manière cartésienne, donc avec goto et en utilisant la liste S résultat de `polyreg`. On évite ainsi de faire de la trigo, mais peut-être voudrez-vous…

• Programmez `plt_polyreg_cart` en remplaçant la tortue par **matplotlib**.

• Modifiez `plt_polyreg_cart` pour qu'elle trace aussi les **rayons** joignant chaque sommet au centre. Chaque rayon aura une **couleur aléatoire.** La liste des couleurs s'obtient en parcourant le texte fourni par : `help(plt.plot)`.

• Avec Matplotlib, **dessinez la courbe** de la fonction $x \mapsto \cos(x) + 3\sin(2x)$. Faites apparaître une grille sur le dessin.

+