shipout/backgroundshipout/foreground

COURS PYTHON TSI1

SOMMAIRE

Chapitre 1

Variables, structured types and loops

Objectifs

- **Les incontournables :**
  - savoir nommer simplement et efficacement une variable et l'affecter ;
  - utiliser `del x` pour désaffecter la variable `x` ;
  - savoir reconnaître les types de base ;
  - savoir utiliser les opérations `not, or, and` sur les booléens ;
  - savoir utiliser les opérations classiques sur les entiers ou sur les flottants ;
  - savoir écrire les comparaisons en code Python ;
  - savoir écrire les différents types structurés : tuple, chaîne et liste ;
  - savoir utiliser les opérations classiques communes à ces types structurés : concaténation, recherche d'un élément, extraction de tranche, répétition etc. ;
  - savoir utiliser les opérations les plus classiques propres aux listes comme `append` ou la méthode `pop` qui permettent d'ajouter ou d'enlever des éléments à une liste ;
  - savoir écrire une instruction conditionnelle avec `if, elif, else` ;
  - savoir utiliser `range` pour créer un intervalle d'entiers ;
  - savoir utiliser une boucle `for` ou une boucle `while` ;
  - savoir utiliser l'instruction `break` ;
  - savoir utiliser la fonction `print` ;
  - savoir utiliser l'instruction `assert`.
- **Et plus si affinités :**
  - Connaître les affectations spécifiques à Python qui permettent l'incrémentation ;
  - savoir écrire une succession croissante d'entiers avec un pas fixé et récupérer ou éliminer des éléments d'une liste avec un pas fixé ;
  - savoir utiliser la fonction `id` :
  - savoir faire un dépaquage de tuples ;
  - savoir utiliser les opérations des listes un peu plus exotiques : `sort, count, insert, sum, del`.

Cours

■ Variables et types

1. Un exemple d'affectation

Commençons par observer un exemple d'affectation :

```
In [1]: var1 = 6+7
In [2]: var1, id(var1), type(var1)

Out[2]: (13, 140707543005344, int)
```

On peut représenter la mémoire de Python en deux zones.

- La table des symboles, où sont stockés les noms des objets. À chaque nom d'objet est associé un type noté **`type`** et une adresse notée **`id`** dans la table des valeurs. L'adresse représente la position de l'objet, elle est repérée par un nombre ;
- La table des valeurs, où sont stockées les valeurs des objets. Ce stockage se fait sous forme binaire (suites de bits valant 0 ou 1).

Ainsi dans l'exemple précédent, la valeur stockée dans la table des valeurs pour **`var1`** est 1101.
Plus précisément, Python procède en deux temps : d'abord il évalue $6 + 7$ puis il stocke le résultat obtenu donc 1101 dans la table des valeurs et note l'adresse utilisée. Enfin, il inscrit le nom **`var1`** dans la table des symboles en précisant le type et l'adresse associés.
Si l'on réaffecte notre variable **`var1`**, c'est-à-dire si l'on lui donne une autre valeur, l'adresse change. En fait, Python supprime le nom de la table des symboles puis le réinscrit, c'est une nouvelle adresse qui est créée.

```
In [4]: var1 = 47*2
In [5]: var1, id(var1)

Out[5]: (94, 140707543007936)
```

2. Les types principaux

Le mieux ce sont encore des exemples.

```
In [6]: var2 = 3.01
In [7]: var2, type(var2)

Out[7]: (3.01, float)

In [8]: var3 = complex(1,2)
In [9]: var3, type(var3)

Out[9]: ((1+2j), complex)
```

```
In [10]: var4 = (3<8)
In [11]: var4, type(var4)
Out[11]: (True, bool)
In [12]: var5 =(3>8)
In [13]: var5, type(var5)
Out[13]: (False, bool)
In [14]: var6 = "tout cela est trop simple "
In [15]: var6, type(var6)
Out[15]: ('tout cela est trop simple ', str)
```

Résumons donc les types rencontrés.

- Le type **`int`** pour les entiers relatifs stockés en valeur exacte ;
- Le type **`float`** pour les nombres à virgule flottante ;
- Le type **`complex`** pour les nombres complexes, retenir que **`j`** est le i des matheux ;
- le type **`bool`** pour les booléens qui renvoie **`True`** ou **`False`** ;
- le type **`str`** pour les chaînes de caractères.

D'autres types existent, par exemple le type **`dict`** : C'est une liste non ordonnée d'objets délimitée par des accolades, appelée dictionnaire. Ce sera vu dans le chapitre 3 et surtout en TSI2.

Revenons aux booléens et les opérateurs associés : **`not`**, **`or`**, **`and`**

```
In [16]: var7 = not(3>=8)
In [17]: var7
Out[17]: True
In [18]: var8 = (4==8) or (4<8) ;  var9 = (4==8) and (4<8)
In [19]: var8, var9
Out[19]: (True, False)
```

3. Les opérations classiques. On a commencé à en voir avec le dernier exemple. **`a >= b`** (respectivement **`a <= b`**) signifie que **`a`** est supérieur ou égal (respectivement inférieur ou égal) à **`b`**. Pour la comparaison, l'égalité entre **`a`** et **`b`** est **`a == b`** et pour signifier que **`a`** n'est pas égal à **`b`**, on écrit **`a != b`**. Le produit de **`a`** et **`b`** se tape **`a*b`** et **`a`** à la puissance **`b`** est **`a ** b`**. Notons que pour écrire une puissance de 10, on peut taper **`1e - 7`** et ce qui donne 10^{-7}. Puis **`max(a,b)`** (respectivement **`min(a,b)`**) renvoie le maximum (respectivement le minimum) entre **`a`** et **`b`**. Enfin la commande

`int(a)` donne l'entier le plus proche de `a`.

4. L'instruction `assert`. Si l'on tape `assert p` où `p` est une proposition, l'exécution envoie un message d'erreur si `p` est fausse.

```
In [22]: assert 3 <= 5

In [23]: assert 5 <= 3
Traceback (most recent call last):
  File "<ipython-input-23-4b731aa5c1c2>", line 1, in <module>
    assert 5 <= 3
AssertionError
```

5. Nom des variables

Le nom d'une variable peut comprendre des lettres, des chiffres et le symbole `_`. Il ne doit pas commencer par un chiffre. Certains mots sont verboten car réservés au langage Python qui peut les utiliser. Par exemple `import`, `True`, `False` etc ou certaines fonctions.

```
In [23]: False = 2023
  File "<ipython-input-23-1ab1d76c5f7b>", line 1
    False = 2023
    ^
SyntaxError: cannot assign to False
```

Notons enfin que Python fait la différence entre les majuscules et les minuscules. On peut ainsi noter une variable `false` mais il vaut mieux éviter ce genre de pirouette.

6. Affectation

La commande d'affectation est le symbole `=` comme on a vu plus haut. Attention donc, `a=8` signifie que l'on affecte la valeur 8 à `a` et `a==8` est une instruction booléene de valeur `True` si 8 est affecté à `a` et de valeur `False` sinon.

```
In [24]: # affectations successives
In [25]: a = 5; b = 3*a
In [26]: a,b
Out[26]: (5, 15)

In [27]: # affectations simultanees
In [28]: a = b = -5
In [29]: a,b,id(a),id(b)
Out[29]: (-5, -5, 140707543004768, 140707543004768)

In [30]: # affectations paralleles
In [31]: a,b,c = 3,4,5
In [32]: c-a+b**2
Out[32]: 18
```

Dans le second cas, on voit que l'on a créé une seule variable et que les noms `a` et `b` renvoient tous les deux vers cette même variable.

7. Suppression d'une variable : On utilise la commande `del`

```
In [75]: x=1
In [76]: x, id(x)
Out[76]: (1, 140707954702112)

In [77]: del x
In [78]: x
Traceback (most recent call last):
  File "<ipython-input-78-6fcf9dfbd479>", line 1, in <module>
NameError: name 'x' is not defined
```

8. Affectations spécifiques à Python qui permettent l'incrémentation

- L'instruction `x += y` est équivalente à `x = x+y` ;
- L'instruction `x -= y` est équivalente à `x = x-y` ;
- L'instruction `x *= y` est équivalente à `x = x*y` ;
- L'instruction `x **= y` est équivalente à `x = x**y` (mathématiquement x^y) ;
- L'instruction `x /= y` est équivalente à `x = x/y` ;

■ Boucles simples

1. Plantons le décor

Un programme est composé d'une suite d'instructions, exécutées l'une après l'autre dans l'ordre où elles sont écrites, contenant des définitions de variables et de fonctions (on verra plus loin des fonctions usuelles), des affectations, des boucles (simples ou imbriquées), des instructions conditionnelles qui utilisent des expressions pouvant être des résultats d'appels de fonctions (l'aboutissement ce sont les fonctions récursives que l'on voit en TSI1 au printemps).

On distingue deux types de boucles (pour l'instant les boucles simples).

- **les boucles conditionnelles**

```
while condition :
    instructions
```

- **les boucles non conditionnelles**

```
for elt in sequence:
    instructions
```

Avec une boucle non conditionnelle, le bloc d'instructions est répété par exemple n fois si n est la longueur de la séquence. Cette séquence peut être une liste, un tuple, une chaîne de caractères). Nous verrons plus loin ce qu'est une liste ou un tuple en Python. La variable `elt` prend la valeur de l'un des n éléments de la séquence. Donnons un exemple.

```
In [9]: for i in range(4):
   ...:     i=i+1
In [10]: i
Out[10]: 4
# VARIANTE qui donne le meme resultat
In [9]: for i in range(4): i=i+1
```

Avec le code qui suit qui utilise une boucle conditionnelle, on aboutit au même résultat.

```
In [11]: i=0
In [12]: while i<4:
    ...:     i=i+1
In [13]: i
Out[13]: 4
```

Comment choisir entre la boucle `for` et la boucle `while` ?

En général, si l'on connait avant de démarrer la boucle le nombre d'intérations à exécuter, on choisit une boucle **`for`**. Au contraire, si la décision d'arrêter la boucle ne peut se faire que par un test, on choisit une boucle **`while`** Nous verrons plus loin des exemples utilisant la boucle appropriée.

2. Quelques créations de boucles simples avec les entiers

❑ **Mode 1.1.—**

- `range(n)` donne les entiers de 0 à $n-1$.
- `range(a,b)` donne les entiers compris entre a (compris) et b (non compris).
- `range(a,b,p)` donne les entiers compris entre a et b avec un pas de p.
- `range(n,i,-1)` si $i < n$, donne les entiers de $i+1$ à n dans l'ordre décroissant.
- `a // b` donne le quotient dans la division euclidienne de a par b.
  Ainsi `a // n` renvoie $\lfloor \frac{a}{n} \rfloor$.
- `a % b` donne le reste dans la division euclidienne de a par b.
  Ainsi `a % b == 0` signifie que a est un multiple de b.
- Pour écrire $S = \sum_{k=1}^{n} a_k$, on peut **utiliser une boucle**.
  Par exemple, pour $n = 5$ et $a_k = k^2$, on tape :

```
S = 1 ** 2
for k in range(2,6) : S += k**2
S
```

  Ou alors on utilise la **fonction prédéfinie** `sum` et dans ce cas, on tape :
  `S = sum(k**2 for k in range(1,6)) ; S`
  Si les a_k sont directement donnés sous forme d'une liste `L` comme on verra plus loin , la commande `sum(L)` donne `S`
- Pour la factorielle de n qui est $n!$, il y a deux méthodes.
  1. Quand on aura vu les modules principaux plus loin, on tapera `from math import factorial` puis on tapera `factorial(n)`
  2. On peut **utiliser une boucle**. Par exemple, pour $n = 5$, on tape :

```
P = 1
for k in range(2,6) : P *= k
P
```

  On obtient $5! = 120$.
- De façon générale, pour écrire $P = \prod_{k=1}^{n} a_k$, on peut utiliser encore une boucle comparable à celle de la somme en remplaçant `S +=` par `P *=`

On peut aller plus loin, par exemple si l'on veut calculer des coefficients binomiaux $\binom{n}{k}$ (qui sont des entiers), on peut le faire en calculant des factorielles et en faisant le rapport, on peut utiliser des fonctions récursives (voir plus tard) et il existe des modules qui les donne directement (par exemple la fonction `binom` de `scipy.special` mais ça ce sera pour ceux qui en veulent plus).

Remarque : Ne pas confondre `a//b` qui est le quotient de la division de a par b et est donc toujours

entier et `a/b` qui est le résultat de la division de a par b et n'est pas toujours un entier.

```
In [21]: 17 // 4
Out[21]: 4
In [22]: 17/4
Out[22]: 4.25
```

3. Instruction conditionnelle

On utilise les instructions prédéfinies en Python qui sont `if`, `elif` et `else` de la façon suivante.

```
# cas d'une alternative

if condition :
    instructions1
else :
    instructions2
```

```
# cas de plusieurs alternatives

if condition1 :
    instructions1
elif condition2 :
    instructions2
elif condition3 :
    instructions3
else :
    instructions4
```

Par exemple :

```
In [28]: x = 10

In [29]: if x < 5 :
    ...:     print(x, "est plus petit que 5 ")
    ...: else :
    ...:     print(x , " est egal ou plus grand que 5")
    ...:
10  est egal ou plus grand que 5
```

4. La fonction prédéfinie `print`

Ce dernier exemple a permis d'introduire une fonction importante de Python qui est `print`. Cette fonction affiche des résultats. On peut en afficher plusieurs en séparant les arguments avec des virgules. Donnons un exemple.

```
In [1]: a=3
In [2]: print(2*a, a*a, a**10)

6 9 59049
```

On voit aussi sur l'exemple du paragraphe précédent que **`print`** peut afficher du texte et la syntaxe est **`print("texte")`**

Exercice 01 *Affecter* 5 *à la variable* **`x`** *puis écrire le code qui affiche « `x` est positif » si `x > 0` et qui affiche « `x` est negatif ou nul » sinon.*

Remarque : Il existe une autre fonction qui agit comme **`print`** c'est-à-dire affiche le résultat. C'est **`return`**. Mais attention, il y a une différence importante. Si l'on fait appel à **`print`** trois fois consécutivement par exemple, il y aura trois affichages et pour **`return`**, c'est le premier exécuté qui s'affiche. On reviendra sur **`return`** plus loin au moment de la construction de procédures Python. Cette fonction sera alors tout à fait adéquate.

5. L'instruction `break`

L'instruction **`break`** permet de casser l'exécution d'une boucle **`while`** ou **`for`**. Elle fait sortir de la boucle et passer à l'instruction suivante éventuelle.

```
In [1]: for i in range(10):
   ...:     print("debut iteration",i)
   ...:     print("good bay")
   ...:     if i == 2:
   ...:         break
   ...:     print("fin iteration",i)
   ...: print("cette horrible boucle est finie")
debut iteration 0
good bay
fin iteration 0
debut iteration 1
good bay
fin iteration 1
debut iteration 2
good bay
cette horrible boucle est finie
```

Notons enfin que dans le cas de boucles imbriquées (voir plus loi dans le cours), l'instruction **`break`** ne fait sortir que de la boucle la plus interne.

6. Les chaînes, les ensembles, les tuples et les listes

Dans tout programme Python, on rencontre rapidement ces types de structures. Il faut les reconnaître et utiliser les bonnes syntaxes.

6-1 Commençons par les tuples

Un tuple permet de créer une **collection ordonnée de plusieurs éléments**. Ce sont mathématiquement des p-uplets. Comme pour les p-uplets, un tuple peut commencer par **`(`** et finir par **`)`**, c'est-à-dire des parenthèses. Mais ce n'est pas obligatoire comme nous allons le voir.

```
In [5]: a = (5,4,7)
In [6]: type(a)
Out[6]: tuple
In [7]: b,c = 5,6
In [8]: b
Out[8]: 5
In [9]: c
Out[9]: 6
In [10]: a = (3,4)
In [11]: u,v = a
In [12]: u
Out[12]: 3
In [13]: v
Out[13]: 4
In [14]: a[0], a[1]
Out[14]: (3, 4)
```

Ainsi, `a[i]` est la valeur d'indice i du tuple `a`.

Remarque : On ne peut pas ajouter, supprimer ou modifier des éléments d'un tuple après sa création, on dit qu'un tuple **n'est pas mutable**.
Les opérations possibles sont : la longueur. Si `a` est le tuple, le code `len(a)` renvoie la longueur du tuple. Puis l'accès par indice positif, la concaténation avec `+`, la répétition avec `*` et la tranche. Nous verrons plus en détail ces opérations avec les chaînes de caractères car c'est la même syntaxe.

Dépaquetage d'un tuple

Il existe quand même une autre opération qu'on peut faire avec l'opérateur `*`.
Le mieux c'est faire un exemple.

```
In [16]: a,b,c="mickey","donald","picsou","daisy"
Traceback (most recent call last):

  File "<ipython-input-16-2ffa94677b58>", line 1, in <module>
    a,b,c="mickey","donald","picsou","daisy"

ValueError: too many values to unpack (expected 3)

# on voit que dans le tuple on attend 3 elements

In [17]: a,*b,c="mickey","donald","picsou","daisy"

In [18]: a,b,c
Out[18]: ('mickey', ['donald', 'picsou'], 'daisy')

# on a alors bien 3 elements car 2 elements
# se sont concatenes en une liste (voir 5-3)
```

6-2 Continuons par les chaînes de caractères

Une chaîne de caractère en Python ou string est une série ordonnée de caractères. Elle peut être une combinaison d'une ou plusieurs lettres, chiffres et caractères spéciaux. Comme pour les tuples, on ne peut pas modifier une chaîne de caractères une fois créée. Elle est **non mutable**. Une chaîne de caractères commencent et finissent par des apostrophes ou des guillemets. Ainsi on a déjà écrit des chaînes de caractères plus haut. elles sont de type `str` et par exemple `"mickey"` est une chaîne de caractères.

Comme pour les tuples, les opérations possibles sont : longueur, accès par indice positif, concaténation avec `+`, répétition avec `*` et tranche. Nous allons donner des exemples détaillés.

Concaténation et répétition des chaînes de caractères en Python

Pour la concaténation :

```
In [1]: str1 = 'Salut!' ; str2 = 'les champions'

# premiere syntaxe

In [2]: print(str1,str2)
Salut! les champions

In [3]: # autre syntaxe

In [4]: print(str1 + '' + str2)
Salut!les champions

In [5]: print(str1 + ' ' + str2)
Salut! les champions

In [6]: # remarquer l'espace entre les apostrophes
```

Pour la répétition :

```
In [7]:  (str1 + ' ' + str2 + " ") * 8

Out[7]: 'Salut! les champions Salut! les champions Salut! les
    champions Salut! les champions Salut! les champions Salut! les
     champions Salut! les champions Salut! les champions '
```

Appartenance et comparaison des chaînes de caractères

L'idée est de tester si une séquence existe ou non dans une chaîne de caractères en utilisant l'instruction `in` de Python.

```
In [14]: str1 = 'Good day'
In [15]: 'b' in str1
Out[15]: False
In [16]: 'b' not in str1
Out[16]: True
In [17]: 'g' in str1
Out[17]: False
In [18]: 'G' in str1
Out[18]: True
In [18]: id('g'), id('G')
Out[18]: (2658593704880, 2658593912112)
```

On remarque comme prévu que **g** et **G** ne sont pas identiques et ne sont donc pas à la même adresse.

Accéder aux caractères d'une chaîne

Nous pouvons accéder aux caractères d'une chaîne en utilisant l'indexation avec des nombres entiers. On rappelle que le premier indice est 0 par défaut et que l'on peut user des indices négatifs. Ainsi l'indice -1 représente le dernier élément et -2 l'avant dernier. Cette technique des indices négatifs est utilisée par exemple si l'on ne connait pas la longueur n de la chaîne. Ainsi l'indice $n-1$ qui correspond au dernier élément est aussi -1.

```
In [21]: str3 = 'python est trop chouette'
In [22]: str3[0] , str3[-1] , str3[6] , str3[23]
Out[22]: ('p', 'e', ' ', 'e')
# remarquer que les espaces dans str3 comptent chacun pour un
    caractere

In [23]: str3[2 : 5]
Out[23]: 'tho'

# On a une commande qui renvoie toute la chaine
In [24]: str3[ : ]
Out[24]: 'python est trop chouette'

In [25]: # on a une commande qui permet de sauter des indices ici
     de indice 1 a indice 8 en sautant 1 indice sur 2
In [26]: str3[2:9:2]
Out[26]: 'to s'
In [27]: str3[0:9:1]
Out[27]: 'python es'
In [28]: str3[::3]
Out[28]: 'ph tr ot'
```

Alors **str[i:j:p]** renvoie de **str[i]** à **str[j-1]** au max en sautant $p-1$ caractères

Exercice 02 *Que renvoie* ***str3[-2]*** *? Que renvoie* ***str3[7 : 11]*** *?*
Quel code pour obtenir **chouette** *en entier ? Que renvoie* ***str3[4:11:4]*** *?*

Caractère d'immuabilité d'une chaîne de caractères. Revenons sur le fait que les chaînes de caractères en Python ne peuvent pas être modifiées. Reprenons l'exemple de `str3` précédent.

```
In [32]: str3[0] = 'P'
Traceback (most recent call last):

  File "<ipython-input-32-7198369e7046>", line 1, in <module>
    str3[0] = 'P'

TypeError: 'str' object does not support item assignment
```

Impossible de remplacer `p` par `P`

6-3 Au tour des listes d'entrer en scène

Elles sont ordonnées comme les tuples et les chaînes de caractères et pour les différencier, elles commencent par le symbole `[` et finissent par le symbole `]`. Et les listes **sont mutables** comme nous allons le voir avec certaines des commandes suivantes. Enfin, on retrouve des opérations communes avec les tuples ou les chaînes de caractères. Par exemple `+` qui concatène deux listes ou $L[i:j]$ qui extrait une partie de la liste. Donnons une liste de ces commandes utiles et on vous conseille de vous référer à cette liste sans modération.

- ***Syntaxe d'une liste.*** Par exemple, on tape `[ 3, 1 , 2 ]`
- ***Liste vide***. Pour la créer, on tape `[ ]`
- ***Longueur d'une liste*** `L`. On tape `len(L)`
- ***Premier élément de la liste*** `L`. On écrit `L[0]`
- ***Dernier élément de la liste*** `L`. On écrit `L[-1]` ou `L[len(L)-1]`
- ***Ajout de l'élément*** `x` en position i dans la liste `L`. On écrit `L.insert(i,x)`
- ***Ajout de l'élément*** `x` en fin de la liste `L`. On écrit `L.append(x)`
- ***Suppression du dernier élément*** `x` de la liste `L`. On écrit `L.pop()`
- ***Suppression de l'élément*** `L[i]` de la liste `L`. On écrit `L.pop(i)`
- ***Renvoie le nombre d'occurences de*** `x` dans `L`. On écrit `L.count(x)`
- ***Somme des éléments d'une liste*** `L`. On tape `sum(L)`
- ***Concaténation de la liste*** `L1` et de la liste `L2`. On écrit `L1+L2`
- ***Suppression de l'élément d'indice*** i dans la liste `L`. On écrit `del(L[i])`
- ***maximum et minimum*** de la liste `L`. On écrit `max(L)` et `min(L)`
- ***Appartenance de l'élément*** `x` à la liste `L`. On écrit `x in L`
- ***Tri d'une liste*** `L`. On écrit `L.sort()`
- ***Création d'une copie d'une liste*** `L` nommée `newL`. On écrit `newL = L[:]`
- ***Sous-liste tirée de*** `L` de i à $j-1$. On écrit `L[i:j]`
- ***Sous-liste tirée de*** `L` ***du début à*** $j-1$. On écrit `L[:j]` ou `L[0:j]`
- ***Sous-liste tirée de*** `L` ***de*** i ***à la fin***. On écrit `L[i:]` ou `L[i:len(L)]`

On peut utiliser les trois derniers items pour les opérations élémentaires sur les lignes d'une matrice. Ce sera fait en TSI2.

Exercice 03 : *Taper le code qui dans une variable* `L` *affecte la liste des entiers pairs de* 0 *à* 10. *Puis écrire l'instruction qui permet d'avoir la sous-liste des deux derniers éléments de* `L` *et écrire l'instruction qui permet de rajouter* 12 *à* `L`. *Enfin, donner l'instruction qui permet d'ajouter* 5 *dans* `L` *pour qu'il soit entre* 4 *et* 6.

Comment créer une liste ?

Il y a plusieurs méthodes. On peut utiliser `append` autant de fois que nécessaire. On peut parfois aller plus vite. Reprenons le cas de l'exercice 03.

```
# Utilisation de la boucle for et de range
In [33]: L = [k for k in range(0,11,2)]
In [34]: L
Out[34]: [0, 2, 4, 6, 8, 10]
```

Exercice 04 : *Créer une liste composé des 10 premiers cubes d'entiers en utilisant une boucle* `for`

On peut aussi créer une liste en répétant un élément.

```
# Utilisation de l'operateur *
In [35]: 14*[0]
Out[35]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Comment parcourir les éléments d'une liste ?

On utilise généralement la boucle `for`

```
In [2]: c = ["Bonnie","Clyde","Laurel","Hardy","Tristan","Iseu"]

In [3]: for i in range(len(c)):
   ...:     print("i vaut", i, "et c[i] vaut", c[i])
   ...:
i vaut 0 et c[i] vaut Bonnie
i vaut 1 et c[i] vaut Clyde
i vaut 2 et c[i] vaut Laurel
i vaut 3 et c[i] vaut Hardy
i vaut 4 et c[i] vaut Tristan
i vaut 5 et c[i] vaut Iseu
```

```
In [4]: for k in c:
   ...:     print("k vaut", k)
   ...:
k vaut Bonnie
k vaut Clyde
k vaut Laurel
k vaut Hardy
k vaut Tristan
k vaut Iseu
```

Comment ajouter ou retrancher des éléments à une liste ?

Pour ajouter en fin de liste, on utilise **`append`**, pour ajouter entre deux éléments, on utilise **`insert`**. Pour enlever le dernier élément de `L`, on utilise **`L.pop()`** et pour enlever l'élément d'indice `i`, on utilisera **`L.pop(i)`**. On peut remarquer que ces opérations transforment définitivement la liste.

```
In [63]: # On cree une liste L puis on enleve son dernier element

In [64]:  L = [1,2,3,4,5,6,7]

In [65]: elt = L.pop()

In [66]: elt, L
Out[66]: (7, [1, 2, 3, 4, 5, 6])

In [67]: # Puis on enleve l'element d'indice 3 qui est 4

In [68]: new_elt = L.pop(3)

In [69]: new_elt, L
Out[69]: (4, [1, 2, 3, 5, 6])

In [70]: # on cherche l'emplacement de L dans la memoire

In [71]: id(L)
Out[71]: 2658740700224

In [72]: # on remet 7 a la fin de L

In [73]: L.append(7)

In [74]: L
Out[74]: [1, 2, 3, 5, 6, 7]

In [75]: # Puis on enleve 7 de nouveau

In [76]: L.pop()
Out[76]: 7

In [77]: L
Out[77]: [1, 2, 3, 5, 6]

In [78]: id(L)
Out[78]: 2658740700224

In [79]: # Conclusion : apres l'action de append et pop(), L se
    retrouve a la meme place dans la memoire qu'avant ces deux
    actions et donc il s'agit de la meme liste.
```

Pour finir le paragraphe des types structurés, on va maintenant mélanger les genres. En effet, les tuples, les listes et les chaînes de caractères peuvent cohabiter dans le meilleur des mondes. C'est l'occasion d'utiliser quelques unes des opérations propres à ces types structurés.

```
In [42]: t1 = 1, 3, 5
In [43]: t2 = "Alfred", 20
In [44]: t3 =("Alfred", 20, ["Trail", "Triathlon"])
In [45]: t4 =("Alfred", 20, ("trail", "triathlon"))

In [46]: t2[0]
Out[46]: 'Alfred'

In [47]: type(t3)
Out[47]: tuple

In [48]: type(t3[2])
Out[48]: list

In [49]: type(t4[2])
Out[49]: tuple

In [50]: t3+t4
Out[50]: ('Alfred', 20, ['Trail', 'Triathlon'], 'Alfred', 20, ('
    trail', 'triathlon'))

In [51]: t3[2]+t4[2]
Traceback (most recent call last):
   File "<ipython-input-51-4cea6a805325>", line 1, in <module>
    t3[2]+t4[2]
TypeError: can only concatenate list (not "tuple") to list

In [52] # on ne peut donc pas concatener un tuple et une liste

In [53]: len(t3), len(t4)
Out[53]: (3, 3)

In [54]: # L'indexation peut fonctionner sur plusieurs niveaux
In [55]: t3[2][1]
Out[55]: 'Triathlon'

In [56]: t3[2][0]
Out[56]: 'Trail'

In [57] # Finissons par plusieurs operations a la fois
In [58]: t1*3+t2
Out[58]: (1, 3, 5, 1, 3, 5, 1, 3, 5, 'Alfred', 20)
```

■ Boucles imbriquées

Terminons ce chapitre par le cas de boucles imbriquées les unes dans les autres. Sous Python, chaque boucle sera à un niveau différent d'indentation. Le mieux comme d'habitude, c'est de faire un exemple.

```
In [79]: S = []
In [80]: for a in range(1,6):
    ...:     for b in range(1,6):
    ...:         S.append([a,b])
```

Que fait cette double boucle ? Au départ, on crée une liste `S` qui est la liste vide. Puis on pose $a = 1$ et on ajoute avec l'attribut `append` dans `L` les listes `[1,b]` où b varie de 1 à 5. Puis on pose $a = 2$ et on continue en ajoutant dans `L` les listes `[2,b]` où b varie de 1 à 5. On s'arrête à $a = 5$ et le dernier élément ajouté est `[5,5]`. On remarque que `S` devient une liste de listes. Vérifions.

```
In [81]: print(S)
[[1, 1], [1, 2], [1, 3], [1, 4], [1, 5], [2, 1], [2, 2], [2, 3],
   [2, 4], [2, 5], [3, 1], [3, 2], [3, 3], [3, 4], [3, 5], [4,
   1], [4, 2], [4, 3], [4, 4], [4, 5], [5, 1], [5, 2], [5, 3],
   [5, 4], [5, 5]]
```

Exercice 05 *Écrire une double boucle utilisant* `for` *et* `range` *qui retourne* $\sum_{i=1}^{3}\sum_{j=1}^{4} ij$.

On pourra appeler `S` *le résultat et initialiser avec* `S = 0`.

Pour finir, l'imbriquement peut concerner des instructions conditionnelles. Par exemple :

```
In [82]: a = 2
In [83]: if a > 0 :
    ...:     if a % 2 == 0 :
    ...:         print("le nombre est positif et pair")
    ...:     else:
    ...:         print("le nombre positif est impair")
    ...:
le nombre est positif et pair
In [84]: a = - 2
In [85]: if a > 0 :
    ...:     if a % 2 == 0 :
    ...:         print("le nombre est positif et pair")
    ...:     else:
    ...:         print("le nombre positif est impair")
In [86]:
```

Dans le second cas, il n'y a rien d'afficher. C'est normal l'instruction `if` intérieure n'est pas sollicitée. Enfin, on peut imbriquer des `while`, des `for`, des `if` etc. Que du bonheur !

Chapitre 2

Introduction to functions and procedures

Objectifs

- **Les incontournables :**
  - connaître la syntaxe pour créer une fonction Python ;
  - connaître la différence entre `return x` et `print(x)` dans une fonction.
- **Et plus si affinités :**
  - savoir faire la différence entre une variable locale et une variable globale dans une procédure ;
  - savoir créer une variable globale ;
  - savoir créer une fonction booléenne.

Cours

■ Définition et création d'une fonction sous Python

1. Définition

Une fonction (on peut dire aussi procédure) est au sens informatique un ensemble d'instructions regroupées sous un nom qui renvoie un résultat lorsque son exécution se termine. Elle représente un sous-programme relativement indépendant du reste du programme et elle peut être appelée en plusieurs endroits du programme. En Python, le type correspondant est **`function`** et la définition d'une fonction doit impérativement respecter les règles ci-dessous.

- La première ligne de la définition commence par le mot-clé **`def`** suivi du nom de la fonction, de parenthèses entourant les paramètres (on dit aussi arguments) de la fonction séparés par des virgules puis du caractère « deux points ». Le nombre d'arguments est variable selon la fonction créée, certaines fonctions n'ont pas d'argument (attention elles conservent quand-même le parenthésage **`()`**)
- Les lignes suivantes (là aussi le nombre de lignes est variable selon la fonction) constituent le bloc d'instructions, indenté par rapport à la ligne de définition et forment le corps de la fonction. Dans ce bloc d'instruction, on peut rencontrer des boucles, des instructions conditionnelles et qui peuvent être imbriquées ;
- Le retour à la ligne signale la fin de la définition de la fonction.

On peut schématiser ainsi :

```
def NOMDEFONCTION(argument1,argument2,...,argumentn):
    BLOC INSTRUCTION
```

Pas de **`begin`** ou de **`end`**. C'est l'indentation des lignes qui permet d'isoler le corps de la fonction.

2. L'instruction `return` et la différence avec `print`

Cette instruction **`return`** est assez fondamentale dans la construction d'une fonction. Elle n'est pas obligatoire (voir par exemple **`ma_fonction`** du paragraphe suivant), mais souvent très utile. En effet son appel dans la procédure permettra de renvoyer non seulement (et d'afficher) le (ou les) résultat(s) voulu(s) mais aussi d'affecter à **`NOMDEFONCTION`** la valeur trouvée.
En effet, supposons avoir construit la fonction **`NOMDEFUNCTION(argument1,argument2)`** alors on donne une valeur à **`argument1`** et à **`argument2`**, respectivement **`v1`** et **`v2`**.
On tape alors **`NOMDEFUNCTION(v1,v2)`** et un résultat doit s'afficher. C'est la commande **`return`** qui va permettre cela.
Donnons un exemple. On crée une fonction de nom : **`multiplie_par_11`** ayant un seul argument **`n`** et qui doit renvoyer $11n$. La commande appropriée est alors **`return 11*n`**

```
In [1]: def multiplie_par_11(n):
   ...:     return 11*n
   ...:
In [2]: multiplie_par_11(5)
Out[2]: 55
```

Remarquons que la syntaxe **`return 11*n`** et **`return(11*n)`** sont équivalentes.

Maintenant, tapons une nouvelle fonction **`new_multiplie_par_11`** qui est identique à la fonction précédente **`multiplie_par_11`** à ceci près que **`return`** est remplacé par **`print`**

```
In [1]: def new_multiplie_par_11(n):
   ...:     print(11*n)
   ...:
In [2]: new_multiplie_par_11(5)
Out[2]: 55
```

On remarque donc que *a priori* les deux fonctions sont comparables. Maintenant tapons :

```
In [5]: multiplie_par_11(7)*2
Out[5]: 154

In [6]: new_multiplie_par_11(7)*2
77
Traceback (most recent call last):

  File "<ipython-input-6-9539eedccad6>", line 1, in <module>
    new_multiplie_par_11(7)*2

TypeError: unsupported operand type(s) for *: 'NoneType' and 'int
   '
```

La différence c'est que **`new_multiplie_par_11(7)`** ne fait que renvoyer l'affichage du résultat qui est 77, c'est un **`NoneType`** pour Python et on ne peut pas le multiplier par 2 qui est un entier.
Conclusion : si seulement afficher le résultat compte, un **`print`** en fin de corps d'instruction suffira et si par contre on doit utiliser le résultat dans d'autres opérations (voire dans d'autres fonctions), un **`return`** s'impose.
Warning : il faut quand même utiliser **`return`** avec modération dans la création d'une fonction. En effet, l'instruction **`return`** interrompt l'exécution de la fonction : aucune des instructions situées après le premier **`return`** ne sera exécutée.
Le mieux encore une fois c'est un exemple pour comprendre et visualiser le problème.

```
In [3]: def multiplie_par_11(n):
   ...:     return ("arretez vos etudes!")
   ...:     return 11*n
   ...:
In [4]: multiplie_par_11(5)
Out[4]: 'arretez vos etudes!'
```

Par contre dans le cas de boucles conditionnelles imbriqués, chaque racine de ces boucles peut contenir un **return** qui ne sera pas en compétition avec les autres. On le verra plus loin dans des fonctions plus élaborées que celles du début du cours.

Exercice 01 : *Écrire une procédure* **MIXAGE_SOM_PROD** *d'arguments* **x** *et* **y** *et qui renvoie un couple de première composante la somme de* **x** *par* **y** *et de seconde composante le produit de* **x** *par* **y**. *On désire ensuite obtenir* **MIXAGE_SOM_PROD(4,2)****3 *Que tape t-on ?*

3. Exemple de fonction sans argument. Introduction aux variables locales et globales

```
In [5]: def ma_fonction():
   ...:     x = 2
   ...:     print("x vaut {x} dans la fonction")
In [6]: ma_fonction()
x vaut 2 dans la fonction
```

Ici **ma_fonction** affichera toujours **x vaut 2 dans la fonction**.
Une remarque intéressante. On affecte la variable **x** avec 2 dans la procédure. Par contre, de retour dans le module principal, il ne connaît plus **x** et renvoie un message d'erreur.

```
In [7]: print(x)
Traceback (most recent call last):

  File "<ipython-input-7-fc17d851ef81>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
```

On dit que la variable **x est locale**.
Remarque : pour éviter des virgules dans l'affichage, on a utilisé la syntaxe **x** entre accolades.

Transformons notre fonction précédente en y mettant **x** comme argument.

```
In [8]: def ma_fonction(x):
   ...:     print("x vaut {x} dans la fonction")

In [9]: ma_fonction(2)
x vaut 2 dans la fonction
```

Encore une fois, on tape **print(x)**

```
In [10]: print(x)
Traceback (most recent call last):
   File "<ipython-input-10-fc17d851ef81>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
```

`x` reste une variable locale.

```
In [13]: def ma_fonction():
    ...:     print(x)
In [14]: x=3
In [15]: ma_fonction()
3
In [16]: print(x)
3
```

Ici comme `x` a été réaffecté hors procédure, elle reste donc visible en dehors de celle là.

```
In [13]: def ma_fonction():
    ...:     print(x)
In [14]: x=3
In [15]: ma_fonction()
3
In [16]: print(x)
3
```

Par contre Python ne permet pas directement la modification d'une variable globale dans le corps de la procédure.

```
In [17]: def ma_fonction():
    ...:     x = x+1
In [18]: x =1
In [19]: ma_fonction()
Traceback (most recent call last):
   File "<ipython-input-19-b20fc4808f49>", line 1, in <module>
    ma_fonction()
   File "<ipython-input-17-227aa57c397b>", line 2, in ma_fonction
    x = x+1
UnboundLocalError: local variable 'x' referenced before
   assignment
```

Il faut indiquer à Python que `x` est bien une variable globale. Et alors ça fonctionne!

```
In [25]: def ma_fonction():
    ...:     global x
    ...:     x=x+1
    ...:     return x
In [26]: x=1
In [27]: ma_fonction()
Out[27]: 2
```

Résumé sur ces deux types de variables

Les **variables locales** sont les variables créées à l'intérieur de la fonction : il s'agit des éventuels arguments et des variables qui sont crééses dans le bloc d'instructions. Ces variables ne sont pas accessibles à l'extérieur de la fonction.

Les **variables globales** regroupent les variables contenant les objets prédéfinis en Python ainsi que les variables créées à l'extérieur de la fonction. Il existe un troisième type de variable globale. Il s'agit de variables créées ou modifiées à l'intérieur d'une fonction. Dans ce cas, on utilise l'instruction `global` comme dans l'exemple précédent.

Warning : une erreur classique est d'utiliser le nom d'une variable globale avec une autre affectation dans la procédure. En effet, quand Python a besoin de connaître la valeur d'une variable `a` par exemple, il cherche d'abord si `a` a une signification localement et ne lui affecte sa valeur globale que s'il n'a pas trouvé de valeur locale. Donnons un exemple.

```
In [7]: a = 5

In [8]: def f(x):
   ...:     a = 2
   ...:     return a*x

In [9]: f(3), a
Out[9]: (6, 5)
```

En effet, pour calculer `f(3)` , Python a utilisé la valeur 2 poue `a` mais quand on lui demande d'afficher `a` à l'extérieur de la procèdure, il se souvient que `a` vaut 5. Allons chercher `global` dans le rôle de Zorro.

```
In [11]: a = 5

In [12]: def f(x):
    ...:     global a
    ...:     a = 2
    ...:     return a*x

In [13]: f(3), a
Out[13]: (6, 2)
```

Python a maintenant mémorisé l'affectation faite à l'intérieur de la fonction.

Conseil. Utiliser l'instruction `global` avec modération. Si l'énoncé vous l'impose, vous n'avez pas le choix. Si vous voulez éviter un souci tel que plus haut, vous créez votre variable globale.
Sinon, en travaillant seulement avec des variables locales, on n'a pas besoin de se préoccuper des noms donnés à l'extérieur de la procédure. Et de plus, si l'on a créé des variables globales, en cas de problème, la détection de ce problème se complique car tout peut être modifié n'importe où. Et enfin, dernier inconvénient, l'encombrement mémoire est bien plus important pour des variables globales que locales car pour ces dernières, la mémoire n'est utilisée que pendant l'exécution de la fonction.

■ Quelques exemples de création de fonctions Pythons

Exemple 01. *Créer une fonction* ***`SOMME_SPECIALE(n)`*** *qui renvoie* $\sum_{i=1}^{n}\sum_{j=1}^{n} i^3 j^2$.

Indication : On pourra introduire une variable locale ***`s`***

Réponse :

```
In [6]: def SOMME_SPECIALE(n):
   ...:     s = 0
   ...:     for i in range(1,n+1):
   ...:         for j in range(1,n+1):
   ...:             s += (i**3)*(j**2)
   ...:     return s
   ...:

In [7]: SOMME_SPECIALE(10)
Out[7]: 1164625

In [8]: SOMME_SPECIALE(0)
Out[8]: 0
```

Exemple 02. *Soit la procèdure :*

```
In [1]: def UNKNOWN01(n):
   ...:     for i in range(20):
   ...:         if (i * n)  % 3 == 0:
   ...:             print(i*n, "*")
   ...:         else :
   ...:             print(i*n)
```

Que fait cette fonction ? Si l'on tape ***`UNKNOWN01(5)`****, que va t-elle afficher ?*

Exemple 03. *Soit la fonction mathématique* g *définie sur* $[0,2[$ *par :*

$$g : x \mapsto \begin{cases} x & \text{pour} \quad 0 \leqslant x < 1 \\ 1 & \text{pour} \quad 1 \leqslant x < 2 \end{cases}$$

Créer une fonction Python nommée **`g`** *d'argument* **`x`** *qui retourne* **`g(x)`** *puis écrire l'instruction qui affiche la liste des valeurs* $g(1/k)$ *pour* $k \in [\![1,6]\!]$.
Si l'on affiche la liste des valeurs $g(k)$ *pour* $k \in [\![1,6]\!]$*, qu'obtient-on ?*

Indication : pour la création de la fonction, on utilisera les instructions conditionnelles **`if`**, **`elif`** *et* **`else`**. *On pourra faire afficher « valeur de l'antecedent erroné » dans le cas où* $x \notin [0,2[$ *dans la procédure. Par ailleurs, on rappelle que* **`a < x and x <= b`** *signifie que* $x \in]a,b]$.

Réponse :

```
In [3]: def g(x):
   ...:     if 0 <= x and x < 1:
   ...:         return x
   ...:     elif 1 <= x and x <2 :
   ...:         return 1
   ...:     else :
   ...:         print("valeur de l'antecedent errone")

In [4]: [g(k) for k in range(1,7)]
valeur de l'antecedent errone
valeur de l'antecedent errone
valeur de l'antecedent errone
valeur de l'antecedent errone
valeur de l'antecedent errone
Out[4]: [1, None, None, None, None, None]

In [5]: [g(1/k) for k in range(1,7)]
Out[5]: [1, 0.5, 0.3333333333333333, 0.25, 0.2,
   0.16666666666666666]
```

Relevons la présence de **`None`** pour signifier qu'il n'y a rien.

Exemple 04. L'intérêt d'une fonction Python c'est qu'elle peut être appelée dans la création d'une autre fonction.

Soit la fonction mathématique f *définie sur* $[0,+\infty[$ *par :*

$$f : x \mapsto \begin{cases} g(x) & \text{pour} \quad 0 \leqslant x < 2 \\ x(x-2) & \text{pour} \quad x \geqslant 2 \end{cases},$$

où g *est la fonction mathématique de l'exemple précédent.*

Créer la fonction Python nommée **`f`** *d'argument* **`x`** *qui retourne* **`f(x)`**.

Indication : encore une fois on utilise les instructions conditionnelles **`if`**, **`elif`** *et* **`else`** *et on pourra faire afficher « valeur de l'antecedent erroné » dans le cas où* $x \notin [0,+\infty[$ *dans la procédure.*

Réponse :

```
In [13]: def f(x):
   ...:     if 0 <= x and x < 2:
   ...:         return g(x)
   ...:     elif x >= 2:
   ...:         return x*(x-2)
   ...:     else :
   ...:         print("valeur de l'antecedent errone")
In [14]: f(1)
Out[14]: 1
In [15]: f(5)
Out[15]: 15
In [16]: f(-1)
valeur de l'antecedent errone
```

Exemple 05. Nous allons créer une fonction booléenne donc qui retourne `True` ou `False`. Ce type de fonction se rencontre fréquemment. Nous allons tester si une liste est un palindrome. Un palindrome se lit indifféremment de gauche à droite ou de droite à gauche en gardant le même sens. Par exemple [*lamarieeiramal*] ou [12321].
Nommons notre fonction `palindrome` d'argument `L` qui renverra `True` si `L` est un palindrome et `False` si `L` ne l'est pas.
Posons donc `L` une liste que l'on veut tester et de longueur `n`, l'idée est de commencer par tester `L[0] != L[n-1]`
Si cette assertion est vraie alors on retourne `False` puis on teste `L[1] != L[n-2]`
Si cette assertion est vraie alors on retourne `False` et on continue ainsi.
On peut donc créer une boucle `for` qui fait varier `i` de 0 jusqu'à non pas $n-1$ (ce serait maladroit !) mais jusqu'à `n//2`.
Enfin imaginons que l'on a jamais `L[i] != L[n-1-i]` Dans ce cas, il faut retourner `True`. On prendra soin de l'écrire après la fin de la boucle `for`. Voilà ce que cela donne :

```
In [17]: def palindrome(L):
    ...:     n = len(L)
    ...:     for i in range(n//2):
    ...:         if L[i] != L[n-1-i] :
    ...:             return False
    ...:     return True
    ...:
In [18]: palindrome([1,2,3,2,1])
Out[18]: True
In [19]: palindrome(["l","a","m","a","r","i","e","e","i","r","a",
   "m","a","l"])
Out[19]: True
In [20]: # on rappelle que les lettres sont des chaines de
   caracteres et mises entre apostrophes ou guillemets

In [21]: palindrome([1,2,3,4,5])
Out[21]: False
```

Exemple 06. Donnons un cas avec beaucoup plus d'arguments. Il s'agit ici d'une procédure qui a 5 arguments : une fonction `f`, trois flottants `t0`, `t1`, `y0` et un entier `n`. Cette procédure renvoie une courbe qui représente une solution d'équation différentielle par la méthode d'Euler. Vous comprendrez les codes Python utilisés dans le bloc d'instruction de cette fonction **`EulerAffich`** plus tard dans l'année et en TSI2.

```
In [22]: def EulerAffich(f,t0,tn,n,y0) :
             h = (tn-t0)/float(n) ; ymin = y0 ; ymax = y0
             t = t0 ;  y = y0 ; T = [t0] ; Y = [y0]
             for k in range(n) :
                 y = y + h*f(y,t) ; t  =  t + h
                 ymin = min(ymin,y)
                 ymax = max(ymax , y)
                 T.append(t) ; Y.append(y)
             deltaT  = (tn-t0)/10 ; deltaY = (ymax-ymin)/10
             plt.plot(T, Y, color = 'b ') ; plt.grid()
             plt.axis([t0-deltaT,tn+deltaT,ymin-deltaY,ymax+
                deltaY])
             plt.axhline(color = ' 0 ')
             plt.axvline(color = ' 0 '); plt.show()
```

Remarque : On peut remarquer que dans le cas de la création d'une fonction un peu longue comme celle là, on peut mettre plusieures commandes indépendantes sur la même ligne de code en les séparant par ;

Exemple 07. Il y a un cas important non encore développé pour l'instant. C'est le cas où la fonction est appelée à l'intérieur de son bloc d'instruction. On parle de fonction récursive. Un chapitre spécial sera consacré bien plus tard dans l'année à ce type de fonction. Il faudra avoir étudié les suites réelles auparavant en Maths. Donnons quand même juste pour le fun une fonction qui s'appelle elle-même pour voir ce que c'est. Reprenons la fonction `g` de l'exemple 03 et transformons la fonction `f` de l'exemple 04.

Soit la fonction mathématique $newf$ définie sur $[0,+\infty[$ par : $x \mapsto \begin{cases} g(x) & \text{si} \quad 0 \leqslant x < 2 \\ f(x-2) & \text{si} \quad x \geqslant 2 \end{cases}$.

La fonction Python nommée **`newf`** d'argument `x` qui retourne **`newf(x)`** pourra s'écrire :

```
In [24]: def newf(x):
    ...:     if 0 <= x and x < 2:
    ...:         return g(x)
    ...:     elif x >= 2 :
    ...:         return newf(x-2)
    ...:     else :
    ...:         print("valeur de l'antecedent errone")
    ...:
```

Faisons fonctionner **`newf`** juste pour voir.

```
In [25]: newf(1)
Out[25]: 1

In [26]: newf(2)
Out[26]: 0

In [27]: newf(3)
Out[27]: 1

In [28]: newf(100)
Out[28]: 0

In [29]: newf(-0.1)
valeur de l'antecedent errone
```

En fait $newf(2n) = 0$ pour tout entier n.

Un autre exemple de fonction récursive que vous rencontrerait en TSI2 concerne le tri fusion qui est le tri le plus performant. Cette année vous verrons d'autres tris, en particulier le tri-bulles dont la programmation ne nécessite pas de récursivité.

Chapitre 3

Search for values. Counting

Objectifs

- **Les incontournables :**
  - savoir créer un compteur simple à partir d'une boucle `for` ou `while` ;
  - savoir créer une variable accumulatrice à partir d'une boucle `for` ou `while` ;
  - savoir créer une fonction booléenne qui indique la présence d'une valeur à partir d'une boucle `for` ou `while` ;
  - savoir rechercher la première occurrence à partir d'une boucle `for` ou `while` ;
  - savoir rechercher la dernière occurrence à partir d'une boucle `for` ou `while` ;
  - savoir créer une fonction qui renvoie le maximum d'une liste ordonnée ;
  - savoir créer une fonction qui renvoie le miniimum d'une liste ordonnée ;
  - savoir reconnaître un dictionnaire.
- **Et plus si affinités :**
  - savoir récupérer les clés ou les valeurs dans un dictionnaire et une valeur connaissant la clé.
  - savoir créer une fonction qui renvoie le nombre de fois où apparaissent tous les éléments d'une liste d'entiers ;
  - savoir créer une fonction qui renvoie le nombre de fois où apparaissent tous les éléments d'une chaîne de caractères ;
  - savoir créer une fonction qui renvoie `True` si un motif appartient à un texte et `False` sinon.

Cours

■ Compteurs

Lorsqu'on utilise une boucle **`for`**, on connaît en général à l'avance le nombre de passage dans la boucle. À moins que l'on ajoute un test (voir l'exemple 2). Mais par contre lorsqu'on utilise une boucle **`while`**, on ne connaît pas ce nombre de passages mais on peut souhaiter compter le nombre de passages dans la boucle. On utilise une variable (en général locale) que l'on peut appeler **`compteur`** ou **`cpt`** par exemple. Cette variable est toujours initialisée à 0 et incrémentée à chaque passage de la boucle. C'est ce que l'on va illustrer par l'exemple 01.

Exemple 01.

On construit un programme **`taille`** qui désire compter le nombre de divisions euclidiennes successives de n par 2, jusqu'à arriver à un quotient nul.

```
In [1]: def taille(n):
   ...:     cpt =0
   ...:     while n > 0:
   ...:         cpt = cpt + 1
   ...:         n = n // 2
   ...:     return cpt
In [2]: taille(2), taille(5), taille(20), taille(180000)
Out[2]: (2, 3, 5, 18)
```

Dans cet exemple, le compteur est incrémenté de toute façon.

Exemple 02.

Le programme suivant va lister les entiers de 1 à n. Et nous incrémentons le compteur chaque fois que cet entier **`d`** est un diviseur de **`n`**, c'est-à-dire si **`n % d == 0`**

```
In [3]: def diviseurs(n):
   ...:     cpt = 0
   ...:     for d in range(1,n+1):
   ...:         if n % d == 0 :
   ...:             cpt = cpt + 1
   ...:     return cpt
In [4]: diviseurs(1),diviseurs(8), diviseurs(1458)
Out[4]: (1, 4, 14)
```

Exercice 01. *Écrire une fonction nommée* **`nombre_de_e`** *d'argument une chaîne de caractères appelée* **`chaine`** *et qui renvoie le nombre de* **`'e'`** *contenus dans* **`chaine`**. *On nommera* **`cpt`** *la variable*

compteur et on appelera `car` *un élément quelconque de* `chaine`*. Pour faire la boucle, on utilisera la syntaxe* `for car in chaine:` *On n'oubliera pas l'instruction conditionnelle.*

```
# Sur un exemple, cela donne :
In [2]: nombre_de_e('en francais, il y a plein de e')
Out[2]: 4
```

Exemple 03. Dans la procédure que l'on construit, on peut avoir à retourner le résultat pour lequel la procédure est créée mais aussi le nombre de passages dans la boucle et donc par exemple il est alors pratique de retourner un couple résultat-compteur. Reprenons diviseurs en ce sens et modifions le.

```
In [2]: def new_diviseurs(n):
   ...:     cpt = 0 ; L = []
   ...:     for d in range(1,n+1):
   ...:         if n % d == 0 :
   ...:             cpt = cpt +1 ; L.append(d)
   ...:     return(cpt, L)
In [3]: new_diviseurs(8), new_diviseurs(458)
Out[3]: ((4, [1, 2, 4, 8]), (4, [1, 2, 229, 458]))
```

■ Accumulateurs

C'est une extension de la notion de compteur. Un accumulateur peut être incrémenté d'une valeur différente de 1 ou même être décrémenté. Le mieux ce sont encore des exemples.

Exemple 01.

```
In [1]: def somme_pairs(L):
   ...:     acc = 0
   ...:     for x in L:
   ...:         if x % 2 == 0 :
   ...:             acc = acc + x
   ...:     return acc
```

Que renvoie `somme_pairs([-2,0,4,1,5,4,8,7])` ?

Exemple 02.

```
In [6]: def mystery01(L):
   ...:     acc = 0
   ...:     for x in L:
   ...:         acc = acc + x
   ...:     acc = acc/len(L)
```

Que renvoie `mystery01([-2,0,4,1,5,4,8,0,1,-1])` ?
Dans la fonction `mystery01`, si n est la longueur de la liste `L`, nous disons que ***le coût est linéaire en*** n car nous opérons n additions et n affectations dans la boucle `for`. Et l'obtention de la longueur par `len(L)` a un coût constant (une seule opération).

■ Recherche de valeurs

1. Recherche de la présence d'une valeur avec `while` ou `for`

Commençons avec des exemples utilisant une boucle `while`. L'idée est de rechercher si `val` est un élément de `tab` qui est un type structuré. Les fonctions proposées seront booléennes c'est-à-dire qu'elles renverront `True` si `val` est dans `tab` et `False` sinon.

Exemple 01. La fonction ici s'appelle `recherche_V1`.
Nommons `presence` la variable booléenne qui est renvoyée à la fin de la procédure. . Elle vaut `False` par défaut.
Comme on utilise `while`, on incrémente avec `i` initialisé à `i=0`. Et on met `i = i+1` dans la boucle en dernière instruction.
Ensuite, que va t-on mettre dans la condition de la boucle ? On doit mettre pour commencer `i < len(tab)` car quand `i` devient au moins aussi grand que `len(tab)`, la quantité `tab[i]` n'a plus de sens. Il faut aussi signaler à Python que dès qu'il découvre que `tab[i]` vaut `val`, il doit sortir de la boucle. On se sert de la variable `presence`. En effet, tant que `val` n'est pas trouvé, la valeur de `presence` reste `False` et `not presence` est alors `True`. Or si l'on rajoute la condition `True` dans une boucle `while`, elle fonctionne et si l'on rajoute par contre la condition `False`, elle ne fonctionne pas. Regardons ce code :

```
In [3]: i = 0
In [4]: while i < 5 and True :
   ...:     i=i+1
In [5]: i
Out[5]: 5
```

Donc la boucle a fonctionné !

```
In [8]: i = 0
In [9]: while i <5 and False :
   ...:     i=i+1
In [10]: i
Out[10]: 0
```

Et ici, la boucle n'a pas fonctionné !
Il est temps de donner le code complet de `recherche_V1`.

```
In [10]: def recherche_V1(tab,val):
    ...:     presence = False
    ...:     i = 0
    ...:     while i < len(tab) and not presence :
    ...:         if tab[i] == val :
    ...:             presence = True
    ...:         i = i+1
    ...:     return presence
```

On fait fonctionner sur des exemples.

```
In [11]: recherche_V1([1,14,0,-1,7],7)
Out[11]: True

In [12]: recherche_V1("je ne sais pas le faire",'f')
Out[12]: True

In [13]: recherche_V1("je ne sais pas le faire",'ff')
Out[13]: False
```

Exemple 02. C'est en fait une forme remaniée de `recherche_V1` que l'on va appeler `recherche_V2`. On n'introduit plus le booléen `presence` et dans la boucle `while`, on remplace `not presence` par `tab[i] != val` et donc on boucle tant que `i<len(tab)` et `tab[i] != val`.

```
In [11]: def recherche_V2(tab,val):
    ...:     i = 0
    ...:     while i < len(tab) and tab[i] != val :
    ...:         i = i+1
    ...:     return tab[i] == val
```

Faisons fonctionner pour voir !

```
In [12]: recherche_V2([1,2,0,4,4,8,4,5],2)
Out[12]: True

In [13]: recherche_V2([1,2,0,4,4,8,4,5],5)
Out[13]: True
```

Tout semble bien aller ! Faisons un cas où `val` n'est pas dans `tab`.

```
In [14]: recherche_V2([1,2,0,4,4,8,4,5],7)
Traceback (most recent call last):
  File "<ipython-input-14-54f3da8e4120>", line 1, in <module>
    recherche_V2([1,2,0,4,4,8,4,5],7)

  File "<ipython-input-11-ef1f0e5bd91e>", line 5, in recherche_V2
    return tab[i] == val
IndexError: list index out of range
```

Zut, on a un problème. En fait l'index **i** est hors du **range**. Cela vient du fait que comme Python ne trouve pas **val** pour le dernier **i**, il incrémente **i=i+1** et **tab[i]** n'existe pas. Donc il faut remplacer la condition **i<len(tab)** par **i<len(tab) -1** et là ça va fonctionner comme on va le voir.

```
In [15]: def recherche_V2(tab,val):
    ...:     i = 0
    ...:     while i < len(tab)-1 and tab[i] != val :
    ...:         i = i+1
    ...:     return tab[i] == val
In [16]:  recherche_V2([1,2,0,4,4,8,4,5],7)
Out[16]: False
```

Question : pourquoi **i <len(tab)** a fonctionné dans **recherche_V1** ? Car pour la dernière valeur de **i** affactée, on ne demande pas d'utiliser **tab[i]** qui n'existera pas si **val** n'a pas été trouvé.

Exemple 03. Maintenant donnons une version avec la boucle **for** qui est entre-nous (légérement) plus simple.

```
In [19]: def recherche_V3(tab,val):
    ...:     for i in range(len(tab)):
    ...:         if tab[i] == val :
    ...:             return True
    ...:     return False
```

L'idée est de retourner **True** la première fois que **tab[i] == val**. Si cela n'arrive pas, on retourne **False**. Ne pas oublier que c'est le premier **return** qui compte et donc si par exemple **tab[0] == val** est vrai, on retourne **True** même si Python continue la boucle avec des valeurs de **tab[i]** différentes de **val**.

```
In [21]: recherche_V3([1,2,0,4,4,8,4,5],7)
Out[21]: False
In [22]: recherche_V3([1,2,0,4,4,8,4,5],1)
Out[22]: True
```

2. Recherche de la première occurence

Exemple 04. Tapons `first_occurrence` qui recherche le premier indice dans `tab` où est `val`. Utilisons une boucle `for` et on inclut le cas où `val` n'est pas dans `tab`.

```
In [22]: def first_occurrence(tab,val):
    ...:     for indice in range(len(tab)):
    ...:         if tab[indice] == val :
    ...:             return indice
    ...:     return (val,"n'est pas present. Achetez des lunettes!")
```

On fait fonctionner.

```
In [23]: first_occurrence([1,4,0,1,1,7,4],4)
Out[23]: 1
In [24]: first_occurrence([1,4,0,1,1,7,4],5)
Out[24]: (5, "n'est pas present. Achetez des lunettes !")
```

3. Recherche de la dernière occurence

Exemple 05. Tapons `last_occurrence` qui recherche le dernier indice dans `tab` où est `val`. Utilisons encore une boucle `for` et on inclut toujours le cas où `val` n'est pas dans `tab`.
L'idée est d'initialiser `indice` à une valeur non entière (par exemple 0.1). Puis on parcourt tout `tab`. Chaque fois que `tab[i]` rencontre `val`, l'indice devient `i`. Et donc si `indice` a été transformé, on retourne la valeur du dernier `indice` qui est le dernier indice où on rencontre `val`. Sinon, on renvoie le même message que pour `first_occurrence`

```
In [27]:  def last_occurrence(tab,val):
    ...:     indice = 0.1
    ...:     for i in range(len(tab)):
    ...:         if tab[i] == val :
    ...:             indice = i
    ...:     if indice != 0.1 : return indice
    ...:     return (val,"n'est pas present. Achetez des lunettes!")
```

On fait fonctionner.

```
In [29]:  last_occurrence([1,4,0,1,1,7,4],4)
Out[29]: 6
In [30]: last_occurrence([1,4,0,1,1,7,4],3)
Out[30]: (3, "n'est pas present. Achetez des lunettes !")
```

4. Recherche d'un extremum

On va rechercher le maximum, le minimum sans utiliser les fonctions prédéfinies `min` et `max`. Attention, il faut avoir une liste d'éléments tous comparables. Donc ici on ne peut pas prendre des caractères. Il faut des entiers ou des flottants.

Cas du maximum

On va appeler `maximum` la procédure et surtout pas `max` qui est prédéfinie. Le but est de retourner le plus grand élément d'une liste `L`. On crée dans la procédure une variable locale `maxi` qui vaut `L[0]` en initialisation. Puis on boucle en partant de `i = 1` et on compare `L[i]` et `maxi`. Si `L[i] > maxi` alors `L[i]` devient le nouveau `maxi`.
On peut remarquer que si deux valeurs de `L` sont égales, on ne tient compte que de la première ce qui peut allèger le nombre d'opérations.

```
In [31]: def maximum(L):
    ...:     maxi = L[0]
    ...:     for i in range(1, len(L)):
    ...:         if L[i] > maxi:
    ...:             maxi = L[i]
    ...:     return maxi
    ...:
In [32]: maximum([-2,0.1,0.01,45,10**2,101,-5])
Out[32]: 101
```

Cas du minimum

Exercice 02. *C'est à vous ! Créer une procédure* `minimum` *qui renvoie le minimum d'une liste* `L` *en s'inspirant de la procédure* `maximum`.

■ Dictionnaire

Un dictionnaire est un objet de type `dict` est une séquence d'association d'une clé et d'une valeur. Une clé peut être n'importe quel objet, un `tuple` mais pas une liste ou un autre dictionnaire. On construit un dictionnaire en associant des successions de couples, chaque couple est constitué d'un premier élément, appelé clé, `keys` en Python et un second appelé valeur, `values` en Python. Les clés ne sont pas mutables, les valeurs associées quelconques (qui peuvent être à leur tour des dictionnaires). Ces clés ne sont pas ordonnées. Pour créer un dictionnaire, on écrit entre des accolades les couples clé-valeur, la clé et la valeur sont séparés par : et les couples par une virgule. Tout ceci sera approfondi d'ailleurs en TSI2. Donnons des exemples.

```
In [1]: d={0:1,'login':'lppr','pass':'lppr','proc':64}
In [2]: d['login']
Out[2]:   'lppr'
In [3]: for cle in d.keys(): print(cle)
Out[3]:  0   login   pass   proc
```

```
In [4]: for val in d.values(): print(val)
Out[4]:  1    lppr   lppr    64

In [5]: for cle,val in d.items(): print(cle,val)
Out[5]:  0 1
         login lppr
         pass lppr
         proc 64
In [6]: valeur = d.pop('proc') ; valeur
Out[6]: 64
```

Ainsi la commande **`for cle in d.keys():`** donne la liste des clés du dictionnaire **`d`** et la commande **`for val in d.values():`** donne la liste des valeurs du dictionnaire **`d`** et si l'on veut tous les couples on tape **`for cle,val in d.items():`**

Donnons maintenant un exemple plus élaboré.

```
In [7]: pays = {"France":{"capitale": "Paris",
                          "population": 68014000,
                          "superficie": 643800.0},
        "Portugal" : {"capitale": "Lisbonne",
                          "population": 10302674,
                          "superficie":92300.0},
        "Italie" :{"capitale": "Rome",
                          "population" : 60359546,
                          "superficie" : 301336.0}}
In [8]: pays["France"]["population"]
Out[8]:   68014000
In [9]:pays["France"]
Out[8]:    {'capitale': 'Paris', 'population': 68014000, 'superficie':
   643800.0}
```

On en a profité pour donner quelques commandes utiles pour les dictionnaires. Retenir que si **`d`** est un dictionnaire alors **`d["nom␣de␣la␣cle"]`** renvoie la valeur de la clé nommée.

Pour finir, donnons des exemples de gros dictionnaires incorporé dans Python.
Donnons l'exemple de **`__builtins__`**. Les clès de ce dictionnaire sont les fonctions classiques (hors modules). Attention à la syntaxe. La barre **`__`** est constituée de deux fois le tiret sous le 8.

```
In [5]: __builtins__.__dict__['min'](5,2,3)
Out[5]: 2
```

Donnons tout ce qu'il y a dans la bête ou plutôt une partie car c'est trop long.

```
In [4]: __builtins__.__dict__
Out[4]:
{'__name__': 'builtins',
 '__doc__': "Built-in functions, exceptions, and other objects.\n
    \nNoteworthy: None is the 'nil' object; Ellipsis represents
     '...' in slices.",
 'abs': <function abs(x, /)>,
 'all': <function all(iterable, /)>,
 'any': <function any(iterable, /)>,
...
 'dir': <function dir>,
 'divmod': <function divmod(x, y, /)>,
 'eval': <function eval(source, globals=None, locals=None, /)>,
...
 'hash': <function hash(obj, /)>,
 'hex': <function hex(number, /)>,
 'id': <function id(obj, /)>,
 'input': <bound method Kernel.raw_input of <spyder_kernels.
    console.kernel.SpyderKernel object at 0x000001A3BFF15700>>,
...
 'max': <function max>,
 'min': <function min>,
 'next': <function next>,
...
 'False': False,
 'True': True,
 'bool': bool,
...
 'dict': dict,
 'enumerate': enumerate,
 'filter': filter,
 'float': float,
...
 'SyntaxWarning': SyntaxWarning,
 'RuntimeWarning': RuntimeWarning,
...
 'cell_count': <function spydercustomize.cell_count(filename=None
    )>,
 '__IPYTHON__': True,
 'display': <function IPython.core.display.display(*objs, include
    =None, exclude=None, metadata=None, transient=None,
    display_id=None, **kwargs)>,
 'get_ipython': <bound method InteractiveShell.get_ipython of <
    ipykernel.zmqshell.ZMQInteractiveShell object at 0
    x000001A3BFF15A00>>}
```

Autre exemple de gros dictionnaire : **`globals()`** à taper en TP d'informatique juste pour le fun.

■ Comptage à l'aide d'un dictionnaire

0. Introduction. L'idée est de considérer un objet écrit sous forme d'un type structuré (liste, chaîne de caractères, tuple...) et de renvoyer les occurences de chaque élément de cet objet (nombre de `e`, nombre de `1` dans l'objet etc.). On généralise les compteurs développés en début de ce chapitre où on s'intéresse à un seul élément en général.

1. Premier cas simple : on compte des entiers naturels

On rentre une liste d'entiers naturels appelée **`entiers`** et on renvoie une liste nommée `L` qui indique le nombre d'occurences de chaque entier entre 0 et **`max(entiers)`**.

```
In [1]: def compte_version01(entiers):
   ...:     m = max(entiers)
   ...:     L = [0 for i in range(m+1)]
   ...:     for n in entiers :
   ...:         L[n] = L[n] + 1
   ...:     return L
```

Exercice 03. *Faire fonctionner à la main* `compte_version01`. *On suppose que* **`entiers`** *est la liste* `[3,8,1,5,7,4,5]`. *Faire fonctionner* `compte_version01` *et retourner les différents* `L` *à chaque étape de la boucle* **`for`**.

Maintenant, faisons bosser Python.

```
In [2]: compte_version01([0,3,4,11,1,1,1,7,4])
Out[2]: [1, 3, 0, 1, 2, 0, 0, 1, 0, 0, 0, 1]
```

```
# on peut prendre des entiers negatifs pour voir !
In [5]: compte_version01([0,-3,-3,-3,9,9,4,11,1,1,1,7,4])
Out[5]: [1, 3, 0, 0, 2, 0, 0, 1, 0, 5, 0, 1]
```

En effet, dans **`entiers`**, si l'on met des entiers négatifs, Python les transforme en entiers positifs car `L[-3]` est aussi `L[12-3]` soit `L[9]`. Et donc -3 et 9 se cumulent dans la liste finale pour les occurrences.

Pour finir, tentons un cas où la liste n'a pas d'entiers.

```
In [9]: compte_version01(['t','r','o','p','c','o','o','l','!'])

Traceback (most recent call last):
  File "<ipython-input-9-49fa973832ff>", line 1, in <module>
    compte_version01(['t','r','o','p','c','o','o','l','!'])
  File "<ipython-input-1-f64fa0b90630>", line 3, in
     compte_version01
    cpts = [0 for i in range(m+1)]
TypeError: can only concatenate str (not "int") to str
```

On voit qu'il faut faire autre chose. D'où la suite.

2. Cas général. On compte des éléments d'une liste qui ne sont pas en général des entiers, ils peuvent être des caractères par exemple. Au lieu de créer une liste cpts, on crée un dictionnaire `d`. En effet, dans un dictionnaire, les éléments sont indéxés par ces clès qui peuvent être de n'importe quel type non mutable comme infofloat, `tuple`. Les valeurs des éléments du tableau sont les clés du dictionnaire.

```
In [6]: def compte_version02(objet):
   ...:     d ={}
   ...:     for elt in objet :
   ...:         if elt in d:
   ...:             d[elt] = d[elt] + 1
   ...:         else:
   ...:             d[elt] = 1
   ...:     return d
```

Le mieux c'est faire fonctionner sur des exemples !

```
# Commencons par une liste d'entiers prise pour compte_version01
In [7]: compte_version02([0,3,4,11,1,1,1,7,4])
Out[7]: {0: 1, 3: 1, 4: 2, 11: 1, 1: 3, 7: 1}
```

```
# Reprenons la liste avec un entier negatif.
In [8]: compte_version02([0,-3,-3,-3,9,9,4,11,1,1,1,7,4])
Out[8]: {0: 1, -3: 3, 9: 2, 4: 2, 11: 1, 1: 3, 7: 1}
```

On voit que là -3 est vraiment pris en compte indépendamment de 9.

```
# Essayons avec une liste de caracteres
In [10]: compte_version02(['t','r','o','p','c','o','o','l','!'])
Out[10]: {'t': 1, 'r': 1, 'o': 3, 'p': 1, 'c': 1, 'l': 1, '!': 1}
```

Et si l'on essaye directement avec une chaîne de caractères, donc un autre type que **`list`**.

```
In [11]: compte_version02('trop cool !')
Out[11]: {'t':1,'r':1,'o': 3,'p':1,' ': 2,'c':1,'l':1,'!':1}
```

On voit que ça marche mais les espaces sont occurencés aussi.
Finissons par une chaîne plus farfelue.

```
In [14]: compte_version02('trop cool ! 1.045 j ecris nimporte
   quoi')
Out[14]: {'t': 2,'r': 3,'o': 5,'p': 2,' ': 7,'c': 2,'l': 1,
 '!': 1, '1': 1, '.': 1,'0': 1,'4': 1,'5': 1, 'j': 1, 'e': 2,
 'i': 3, 's': 1, 'n': 1,'m': 1, 'q': 1,'u': 1}
```

■ Recherche textuelle

Il s'agit maintenant de rechercher la présence ou l'absence d'un motif ou d'un mot dans un texte. Le texte ou le motif est représenté en Python par des chaînes de caractères (type **`str`**). Ils sont donc composés de caractères qui peuvent être des lettres, des signes de ponctuations, des symboles ou de l'espace. Si le motif est constitué d'un seul caractère, on est ramené au paragraphe de recherche de valeurs de ce chapitre et aux fonctions **`recherche_V1`** ou **`recherche_V2`**. Dans ce cas, on sait que le coût de la recherche est linéaire de valeur la longueur de la chaîne.

Nous allons donc construire une procédure nommée **`recherche_motif`** qui va être booléenne.
On renvoie **`True`** si **`motif`** est bien dans **`texte`** et **`False`** sinon. Donnons la procédure :

```
In [7]: def recherche_motif(texte,motif):
   ...:     n = len(texte) ; m = len(motif)
   ...:     for j in range(n-m+1):
   ...:         i = 0
   ...:         while i < m and texte[j+i] == motif[i]:
   ...:             i = i + 1
   ...:         if i == m :
   ...:             return True
   ...:     return False
```

Exercice 04. *Faire tourner à la main* **`recherche_motif`** *pour* **texte = "premier essai"** *et* **motif = "er es"** *puis* **motif = "e es"**

```
In [8]: recherche_motif("premier essai", "er es")
Out[8]: True

In [9]: recherche_motif("premier essai", "e es")
Out[9]: False

# Un autre texte pour le fun !
In [10]: recherche_motif("on recherche super bien!", "super")
Out[10]: True
In [11]: recherche_motif("on recherche super bien!", "????")
Out[11]: False
```

Faisons une variante où on renvoie soit le premier indice de texte où motif apparaît, soit on renvoie que ce motif n'est pas dans le texte.

```
In [12]: def recherche_motif_V2(texte,motif):
   ...:     n = len(texte) ; m = len(motif)
   ...:     for j in range(n-m+1):
   ...:         i = 0
   ...:         while i < m and texte[j+i] == motif[i]:
   ...:             i = i + 1
   ...:         if i == m :
   ...:             return j,' est premier indice de motif dans texte'
    ...:     return 'aucune inclusion de ce motif dans le texte'

In [13]: recherche_motif_V2("on recherche super bien!", "????")
Out[13]: 'aucune inclusion de ce motif dans le texte'

In [14]: recherche_motif_V2("on recherche super bien!", "super")
Out[14]: (13, 'est premier indice de motif dans texte')
```

Pour finir, parlons du coût. Notons n la longueur du texte et m celle du motif. La boucle **`for`** est parcourue $(n-m+1)$ fois. Dans le pire des cas, la boucle interne **`while`** est parcourue au plus m fois pour tester chaque caractère du motif. Le nombre total de comparaisons entre caractères est donc majoré par $m(n-m+1)$.

Chapitre 4

The bubble sort

Objectifs

- **Les incontournables :**
  - savoir faire tourner le tri-bulles sur une liste donnée.
- **Et plus si affinités :**
  - savoir utiliser une variable booléenne qui permette de sortir de la boucle d'un tri-bulles s'il n'y a pas eu de permutations à un parcours donné.

Cours

■ Introduction

Un problème classique est le triage de données. Il faut que les données soient comparables les unes aux autres. Le cas le plus simple est une liste d'entiers par exemple et la liste `[1,5,2,4,3,1]` peut se trier selon l'ordre croissant et donner `[1,1,2,3,4,5]` ou selon l'ordre décroissant `[5,4,3,2,1,1]`. On peut remarquer dans cet exemple la présence de la même valeur deux fois. Certains tris vont permuter les deux `1` dans leurs exécutions et certains tris ne vont pas les permuter. On parle de tris stables ou de tris instables. Parmi les tris classiques stables, on peut citer le tri par insertion, le tri bulle (celui qui va nous intéresser dans ce chapitre), le tri Shaker, le tri Gnome et le tri fusion. Parmi les tris classiques instables, le tri par sélection, le tri à peigne, le tri rapide, le tri Shell et le tri maximier. Les longs week end pluvieux, vous pourrez vous pencher sur ces différents tris. Lequel choisir ? Ce qui est le plus important pour un tri c'est sa complexité qui est en rapport avec son temps d'exécution. Sachez que le meilleur de tous est le tri fusion. Nous reparlerons de la plupart de ces tris dans des chapitres ultérieurs. Maintenant passons au tri à bulles qui est celui qu'on va développer dans ce chapitre.

■ Tri à bulle

L'algorithme du tri à bulle consiste à parcourir une liste plusieurs fois jusqu'à ce qu'elle soit **triée du plus petit au plus grand**, en comparant à chaque parcours les éléments consécutifs et en procédant à leur échange s'ils sont mal triés.

Les étapes sont :

- on parcourt la liste et si deux éléments consécutifs sont rangés dans le désordre, on les échange ;
- si à la fin du parcours au moins un échange a eu lieu, on recommence l'opération ;
- sinon, la liste est triée, on arrête.

Notons que si les deux éléments sont identiques, aucune permutation n'est effectuée.
Les éléments les plus grands remontent ainsi dans la liste comme des bulles d'airs qui remontent à la surface d'un liquide. D'où le nom de tri-bulle.
Le mieux maintenant c'est un exemple. Soit la liste $L = [12,17,14,11,8]$. On écrira en gras les éléments permutés.

Parcours 1 : [12,**14,17**,11,8] puis [12,14,**11,17**,,8] puis [12,14,11,**8,17**]
Parcours 2 : [12,**11,14**, 8,17] puis [12,11, **8,14**,17]
Parcours 3 : [**11,12**, 8,14,17] puis [11, **8,12**, 14,17].
Parcours 4 : [**8,11**, 12,14,17].
Parcours 5 : [8,11,12,14,17].

Lors du parcours 5, aucune permutation n'est effectuée. La liste est donc triée.
Remarquons qu'au bout du parcours 1, le plus grand élément 17 est en fin de liste. Au bout du parcours 2, les deux plus grands éléments $14, 17$ sont en fin de liste. Au bout du parcours 3, les trois plus grands éléments $12, 14, 17$ sont en fin de liste. Et ensuite en fin de parcours 4, les quatre

plus grands $11, 12, 14, 17$ sont en fin de liste. Et ici le plus petit (le cinquième élément) est déjà à sa place tout à gauche de la liste.

Exercice 01 *Refaire le tri à bulle à la main sur la liste* $[2, -1, 0, 4, 5, -1, 1]$

Cas le plus favorable pour un tri-bulle. Supposons une liste de n éléments déjà triée :

$$[1, 2, ..., n-1, n].$$

On effectue les $n-1$ comparaisons de 1 avec 2 puis 2 avec 3 etc. et aucune permutation n'est nécessaire. C'est le cas le plus favorable.

Cas le plus défavorable pour un tri-bulle. Supposons que notre liste de départ soit

$$[n, n-1, n-2, ..., 3, 2, 1].$$

La valeur n va se déplacer après chaque itération jusqu'à la fin de la liste. Donc le nombre de comparaisons puis de permutations pour la valeur n est $n-1$.
De manière similaire, pour amener $n-1$ à la bonne place, il faut $n-2$ permutations (le dernier élément est n et donc $n-1$ ne permutera pas avec lui.
Puis pour amener $n-2$ à la bonne place, il faut $n-3$ permutations (les deux derniers éléments sont $n-1$ et n et ne sont pas touchés).
Ainsi de suite... jusqu'à $[2, 1, 3, 4, ..., n-1, n]$. L'élément 2 est juste comparé et permuté avec 1. Et ensuite on parcourt toute la liste une fois et on constate que plus rien ne permute.
Combien d'opérations avons nous fait ?
Pour les comparaisons et les permutations,

$$2\left((n-1) + (n-2) + ... + 1\right) = 2 \times \frac{n(n-1)}{2} = n(n-1).$$

Donc le nombre d'opérations est de l'ordre de grandeur de n^2.
Pour définir la complexité d'un tri, on détermine celle du cas le plus défavorable. On dit ici que **la complexité de ce tri est quadratique**.

Remarque : il y a des tris plus performants. Par exemple la complexité du tri fusion est en $n \ln n$.

■ Mise en place du programme Python

Pour commencer, rappelons une commande utile pour la suite.

```
In [1]: a = 5
In [2]: b = 7
In [3]: a,b = b,a
In [4]: a
Out[4]: 7
In [5]: b
Out[5]: 5
```

Le code `a,b = b,a` permute `a` et `b`

Nous allons créer maintenant le programme `tri_bulles` d'argument `L` en se servant de la remarque qui suit l'exemple [12,17,14,11,8]. On procède à de moins en moins de comparaisons dans l'algorithme. On utilise donc une boucle rétrograde avec le code `range(b,a,-1)` qui liste les entiers de `b` à `a+1` avec `b` strictement supérieur à `a`

```
In [6]: def tri_bulles(L):
   ...:     n = len(L)
   ...:     for dernier in range(n-1,0,-1):
   ...:         for i in range(dernier):
   ...:             if L[i] > L[i+1]:
   ...:                 L[i],L[i+1] = L[i+1],L[i]
   ...:     return L

In [7]: tri_bulles([12,17,14,11,8])
Out[7]: [8, 11, 12, 14, 17]
```

Exercice 02 : *faire fonctionner à la main* `tri_bulles` *avec* `L = [3,0,-2,5,1]`

Revenons à l'algorithme principal. Ce tri ne permute `L[i]` et `L[i+1]` que si `L[i] > L[i+1]` et donc deux éléments de même valeur peuvent par des permutations successives se rapprocher et se retrouver côte à côte mais ne seront pas permutés. Le tri est donc bien stable. Si l'on remplace dans `tri_bulles` par `if L[i] >= L[i+1]`, on obtient le même résultat mais il peut y avoir davantage de permutations et le tri ne serait plus stable.

■ Une version améliorée de Tri-bulles

On va s'inspirer de `recherche_V1` du chapitre 3. Nous introduisons dans `tri_bulles` une variable booléenne `permut` indiquant lors de la fin d'un parcours si l'on a effectué une permutation. Si ce n'est pas le cas le tableau est trié et on peut sortir de la boucle qui fait les comparaisons.
Cette variable doit être réinitialisée à `False` à chaque début d'un parcours (car avant de commencer un parcours, on n'a encore réalisé aucune permutation sur ce parcours).
Par ailleurs, ici la boucle `for` devient une boucle `while` car plus adaptée. Ce qui permet dans la condition de `while` de mettre le test d'arrêt suivant : si `permut` reste `False` donc il n'y a eu aucune comparaison, alors la boucle `while` ne fonctionne plus et on retourne `L`.
On tape donc `while dernier >= 1 and permut:` et la boucle ne fonctionnera tant que `dernier` n'est pas nul et tant que `permut` est vraie.
Enfin, `permut` est initialisé à `True` car sinon on n'effectuera aucun parcours.

```
In [9]: def new_tri_bulles(L):
   ...:     n = len(L)
   ...:     dernier = n-1
   ...:     permut = True
   ...:     while dernier >= 1 and permut:
   ...:         permut = False
   ...:         for i in range(dernier) :
   ...:             if L[i] > L[i+1]:
   ...:                 L[i],L[i+1] = L[i+1], L[i]
   ...:                 permut = True
   ...:         dernier -= 1
   ...:     return L
```

Chapitre 5

Modules and library

Objectifs

- **Les incontournables :**
  - Connaître le nom des modules ou bibliothéques les plus utiles qui sont : `numpy`, `numpy.linalg`, `matplotlib.pyplot` et `random`
  - Savoir créer un alias pour un module ou une bibliothéque (par exemple `import matplotlib.pyplot as plt`).
  - Savoir utiliser la commande `dir`.
  - Connaître la différence entre importer tout un module (par exemple `from math import *`) et certaines fonctions d'un module (par exemple `from math import sin, cos`)
  - Comment nommer une fonction d'un module importé par son nom (par exemple `rd.randint` appelle la fonction `randint` du module `random` ramené à son alias `rd`).
- **Et plus si affinités :**
  - Connaître le module `time` et sa fonction `perf_counter` qui affiche le temps d'exécution.
  - Connaître le module `sympy` qui permet le calcul formel.
  - Connaître la bibliothéque `scipy` et son module `scipy.stats` qui fournit les lois classiques de probabilité.

Cours

■ Introduction

Il est conseillé de décomposer un programme en particulier à l'aide de fonctions. Lorsque des fonctions traitent toutes d'un même sujet, on peut les regrouper dans un fichier, un module. Différents modules peuvent être regroupés dans une bibliothèque. Un programme en Python commence souvent par des lignes contenant le mot **`import`** afin d'utiliser des modules et des bibliothèques (**`library`** en langage Python).

■ Création d'un module

Pour créer un module qui est un simple fichier, il suffit donc d'écrire les définitions de fonctions et de constantes dans un fichier dont le nom a l'extension **`py`**. Il est important de fournir une documentation qui indique la façon de les utiliser et ce qu'elles produisent.

Exemple 01. Nous allons créer un fichier qui se nomme **`aires.py`**, fichier dans lequel on précise la valeur utilisée pour le nombre π et les définitions de quelques fonctions permettant de calculer des aires.
On tape dans l'éditeur :

```
pi = 3.14159
def disque(rayon):
        """rayon de type float est le rayon d'un disque"""
        """et renvoie l'aire du disque """
        return pi*rayon*rayon
def rectangle(largeur,longueur):
        """largeur et longueur sont de type float"""
        """renvoie aire rectangle de cotes largeur et longueur"""
        return largeur * longueur
def triangle(base,hauteur):
        """base et hauteur de type float sont la base et la
           hauteur d'un triangle"""
        """renvoie l'aire du triangle"""
        return base*hauteur/2
```

On nomme ce fichier **`aires.py`**. On a ainsi créer un module appelé **`aires`**. Puis on sort de ce fichier et on se met dans un nouveau fichier vierge de l'éditeur. Puis toujours dans l'éditeur, on tape :

```
import aires
# On appelle ainsi le module aires
print(aires.disque(10))
print(aires.rectangle(30,4))
```

Puis on clique sur **run** et alors dans la console apparaît :

```
In[1] : runfile('C:/Users/damin/.spyder-py3/untitled0.py', wdir='
   C:/Users/damin/.spyder-py3')
Reloaded modules: aires
314.159
120
```

Le point entre **aires** et **disques** signifie que l'on se refère au nom **disque** qui est défini dans le module **aires**.

On accède à la spécification d'une fonction avec la fonction **help**.

```
In [2]: help(aires.disque)
Help on function disque in module aires:

disque(rayon)
    "rayon de type float est le rayon d'un disque
```

Le contenu de tout le module **aires** s'obtient avec la fonction **dir**

```
In[3]: dir(aires)
Out[3]:
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'aires',
 'disque',
 'pi',
 'rectangle',
 'triangle']
```

■ Résumé des commandes pour manipuler les modules

Comment charger un module ou une fonction d'un module ?

Soit un module générique que nous appelerons **package**. Il contient toutes les expressions et fonctions python définis dans le fichier **package.py**

Pour importer **package** ou des fonctions ou des classes incluses dans **package**, on applique une des règles suivantes.

import package : accès à la fonction **fonction** de **package** en tapant **package.fonction**

import package as pa : permet de définir l'alias **pa** de **package**. Ainsi on tape maintenant

`pa.fonction` pour aller chercher `fonction` du module `package`.
`from package import *` : permet de charger toutes les fonctions de `package` et pour aller chercher `fonction`, on ne tapera plus `pa` ou infopackage devant.
`from package import fonction` : ne charge que `fonction` et pas le reste du module `package`.
Par extension, `from package import fonction1 , fonction2 , fonction3` : chargera les trois fonctions citées de `package` et rien d'autre.
Parfois la fonction qui nous intéresse est dans un `sous-package` d'un module principal `package`
On tape `import package.sous-package as sp` ou `from package.sous-package import fonction` par exemple pour l'utiliser.

Comment s'informer sur le contenu d'un module ou sur une fonction d'un module ?
Comme on a vu plus haut sur l'exemple 01, `dir` liste les modules ou fonctions déjà chargés à un moment donné.
`dir(package)` liste les fonctions et constantes du module `package`.
infohelp(package) renvoie des informations sur les fonctions du module `package`
`help(package.fonction)` renvoie des informations sur `fonction` du module `package`.

Donnons maintenant les modules incontournables auxquels on en ajoute quelques uns qui sont intéressants pour que le candidat aux concours soit plus efficace et plus rapide.

■ Une liste de modules ou bibliothéques classiques prédéfinis par Python

Comment différencier un module ou une bibliothéque ? En fait, une bibliothéque est une collection de modules. On les reconnaît car on peut atteindre des modules que ces bibliothèques contiennent avec un attribut. Ainsi si `BIBLIO` est la bibliothèque, `BIBLIO.MOD1` est le module `MOD1` inclus dans `BIBLIO`. À l'appel de `BIBLIO.MOD1`, on peut alors récupérer des fonctions de ce module.
Enfin, `BIBLIO` peut contenir directement des fonctions et être considéré comme un de ses modules.

1. La bibliothèque numpy

La bibliothèque `numpy` permet de faire du calcul scientifique et de manipuler des tableaux. C'est la library que l'on utilisera le plus. Son alias usuel est `np`.
Le module de `numpy` utilisé principalement est `linalg`, consacré à tout ce qui est calcul matriciel.
On l'importe avec : `import numpy.linalg as alg` et donc son alias usuel est `alg`

Exemple 02 *Utilisons* ***numpy*** *pour résoudre :* $\begin{cases} 10^{-20}x + y &= 1 \\ x + y &= 2 \end{cases}$.

```
In[1] : import numpy as np ; import numpy.linalg as alg

In[2] : A = np.array([[10**(-20), 1],[1,1]])
In[3] : B =  np.array([[1],[2]])

In[4] : alg.solve(A,B)

Out[4] :
array([[1.],[1.]])
```

Notons aussi le module `polynomial`, utile comme son nom l'indique, pour le calcul polynomial. On l'importe avec `from numpy.polynomial import Polynomial` et donc son alias usuel est `Polynomial`.

2. Le module math

Le module **math** contient les principales fonctions et constantes mathématiques usuelles. Son alias classique est **mt** et rappelons que l'on peut obtenir des informations sur une fonction particulière à l'aide de la fonction **help**

Exemple 03 Parmi les fonctions de **math**, on remarque la fonction **hypot**. Que fait cette fonction ? Utilisons **help**

```
In [9]: help(math.hypot)
Traceback (most recent call last):
   File "<ipython-input-9-f28b798b5134>", line 1, in <module>
    help(math.hypot)
NameError: name 'math' is not defined
In [10]: # Normal il n'y a pas le module math. Je suis IDIOT !
In [11]: import math
In [12]: help(math.hypot)
Help on built-in function hypot in module math:

hypot(...)
    hypot(*coordinates) -> value

    Multidimensional Euclidean distance from the origin to a
       point.

    Roughly equivalent to:
        sqrt(sum(x**2 for x in coordinates))

    For a two dimensional point (x, y), gives the hypotenuse
    using the Pythagorean theorem:  sqrt(x*x + y*y).

    For example, the hypotenuse of a 3/4/5 right triangle is:
       >>> hypot(3.0, 4.0)
        5.0
```

On a compris, la fonction **hypot(x,y)** renvoie la longueur de l'hypothènuse sachant que x et y sont les longueurs des côtés adjacent et opposé d'un triangle rectangle.

3. Le module cmath

Le module **cmath** permet d'utiliser certaines commandes relatives aux nombres complexes. Il possède beaucoup de fonctions en commun avec **math**. À vous de fouiller (un jour pluvieux).

4. Le module sympy

Le module **sympy** permet de faire du calcul formel, c'est-à-dire de manipuler des symboles et des expressions littérales, sans utiliser des valeurs numériques. C'est un module qui mérite d'être connu car il peut être très utile par exemple pour avoir l'expression littérale d'une dérivée ou d'une primitive exacte. À utiliser pour vérifier un calcul. On peut même l'utiliser pour résoudre des systèmes linéaires à paramètre (chronophage dans une colle de maths donc si la machine le fait pour vous c'est mieux).

Exemple 04. *Utilisons* **sympy** *pour résoudre* $(m \in \mathbb{R})$: $\left\{ \begin{array}{rcl} x+my+z & = & 1 \\ x+y+mz & = & 0 \\ 7x+(4+m)y+(3+2m)z & = & -1 \end{array} \right.$.

```
In[1] : from sympy import *
In[2] : m = symbols('m'); x = symbols('x'); y = symbols('y'); z
    = symbols('z')
In[3] : syst = Matrix([[1,m,1,1],[1,1,m,0],[7,4+m,3+2*m,-1]])
In[4] : solve_linear_system(syst,x,y,z)
Out[4] :
 { z : (m-2)/(2*m*(m-1)), x : -(m+2)/(2*m) ,
   y : (3*m-2)/(2*m*(m-1)) }
```

On trouve les solutions exactes en fonction de m (mais on occulte le cas $m = 0$ ou $m = 1$).

$$x = \frac{-(m+2)}{2m},\ y = \frac{3m-2}{2m(m-1)},\ z = \frac{m-2}{2m(m-1)}.$$

Exemple 05 *On veut calculer* $\int_0^1 e^{-x}\,dx$ *en utilisant* **integrate** *de* **sympy**

```
In[5] : from sympy import *
In[6] : x = symbols('x'); integrate(exp(-x),(x,0,1))
Out[6] :
   - exp(-1)+1
```

5. La bibilothèque scipy

La bibliothèque **scipy** permet elle aussi de faire du calcul scientifique. On utilise principalement deux de ses modules. On les importe en utilisant leurs alias. On tape **import scipy.optimize as resol** et **import scipy.integrate as integr**

Exemple 05 bis *On veut calculer* $\int_0^1 e^{-x}\,dx$ *en utilisant* **quad** *de* **scipy.integrate** *défini par son alias* **integr**

```
In[7] : import math as mt; import scipy.integrate as integr
In[8] : def f(t) : return mt.exp(-t)
In[9] : integr.quad(f,0,1)
Out[9] :
   (0.6321205588285578 , 7.017947987503856e-15)
```

On remarque que l'on obtient deux flottants, le premier correspond à une valeur approchée de $-e^{-1}+1$ et le second une majoration de l'erreur commise en prenant cette valeur approchée à la place de $-e^{-1}+1$.
Ainsi avec **sympy** on a de l'intégration formelle c'est-à-dire que l'on trouve la valeur exacte (si bien entendu c'est possible) et avec **scipy**, on a de l'intégration numérique.

D'ailleurs un TP d'info sera consacré plus tard à l'intégration numérique et notamment aux formules des rectangles ou des trapèzes.

6. Le module Turtle

C'est un module amusant. Il sert par exemple à dessiner des fractals notamment la dragon curve (courbe du dragon) ou le Schneeflocke (flocon de neige) de Von Koch qui sont des répétitions de motifs géométriques et seule la capacité de la machine nous arrête dans l'élaboration du dessin. Ce sont des programmes récursifs donc on ne va pas s'y attarder dans ce chapitre. Donnons simplement un exemple de manipulation de ce module.

Exemple 06. *Comment dessiner un pentagone avec* **`turtle`** *?*
On prendra **`longcot`** *pour mesure d'un côté.*

```
In [1]: from turtle import *

In [2]: def pentagone(longcot):
   ...:     for i in range(5):
   ...:         forward(longcot)
   ...:         left(72)
In [3]: pentagone(200)
```

On commence par appeler le module **`turtle`** puis on commence par faire un premier côté de longueur **`longcot`** (on prendra 200 pour valeur numérique). C'est la fonction **`forward(longcot)`**. Puis on tourne de 72 degrés vers la gauche. C'est la commande **`left(72)`**. Pourquoi 72 ? Car $72 * 5 = 360$. On fait cela 5 fois et on obtient le pentagone voulu qui se place dans une fenêtre de Python Turtle Graphics.
Donnons deux autres commandes classiques (pas utiles pour le pentagone). Par exemple **`right(30)`** va déplacer le curseur de 30 degrés vers la droite, **`backward(150)`** va faire reculer le curseur de 150. Il y a plein d'autres fonctions (pour créer de la couleur, augmenter ou dimimuer l'épaisseur du trait, pour cacher la tortue ou la montrer etc.)

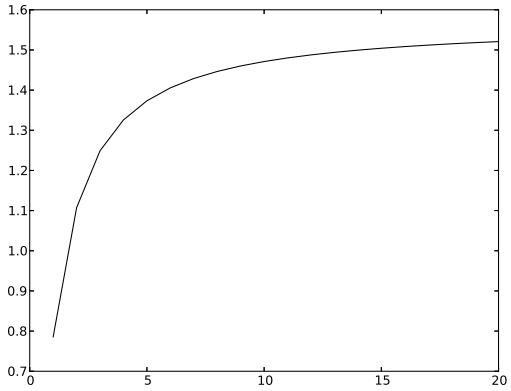
7. La bibliothèque matplotlib

La bibliothèque **`matplotlib`** permet de faire du graphisme (tracé de courbes en particulier). C'est la bibliothèque la plus importante à connaître avec **`numpy`**. On importe son module principal important **`pyplot`** avec l'instruction : **`import matplotlib.pyplot as plt`** et l'alias est donc **`plt`**.

Exemple 07. *Écrire les lignes Python qui permettent de tracer la ligne de segments brisés passant par les points* $(k, f(k))$*, où* k *varie de* 1 *à* 20*. Appliquer et afficher avec* $f : x \mapsto \arctan x$*.*

```
In[4] : import matplotlib.pyplot as plt ; import numpy as np
In[5] : def f(x) : return(np.arctan(x))
In[6] : X = [ k  for k in range(1,21) ]
In[7] : Y = [ f(k) for k in range(1,21) ]
In[8] : plt.plot( X,Y,color = '0'); plt.show( )
```

On obtient la courbe suivante.



8. La bibliothèque random

La bibliothèque **random** permet en fait de générer des nombres aléatoires. On l'importe en totalité avec : **import random as rd** pour utiliser ses fonctions directement accessibles.

- **rd.randint(a,b)** permet de choisir un entier au hasard dans $[\![a, b-1]\!]$.
- **rd.random( )** renvoie un réel dans $[0, 1[$.

Exemple 08. *Création d'une variable de Bernoulli de paramètre p.*
Écrire une fonction **lancer(p)** *qui renvoie* **True** *avec une probabilité* $p \in]0, 1[$ *et* **False** *avec une probabilité* $q = 1 - p$. *On utilisera la fonction* **rd.random( )** *et faire quelques essais.*

```
In [16]: import random as rd
In [17]: def lancer(p):
    ...:     return rd.random()<p
    ...:
In [18]: lancer(0.9),lancer(0.9999),lancer(0.0001)
Out[18]: (True, True, False)
In [19]: lancer(0.4),lancer(0.4),lancer(0.4)
Out[19]: (True, True, True)
In [20]: lancer(0.4),lancer(0.4),lancer(0.4)
Out[20]: (False, False, True)
```

Il faut savoir que **numpy** en tant que bibliothéque possède un module **random** que l'on nommera donc **numpy.random** qui n'est pas exactement le même module que **random** ici. Le module **numpy.random** de **numpy** permet d'aller plus loin. Ainsi, la fonction **randint** de **numpy.random** possède un troisième argument qui permet la répétition.

```
In[21]: import numpy.random as rd
In[22]: rd.randint(0,2,4)
Out[22]: array([0, 1, 1, 1])
```

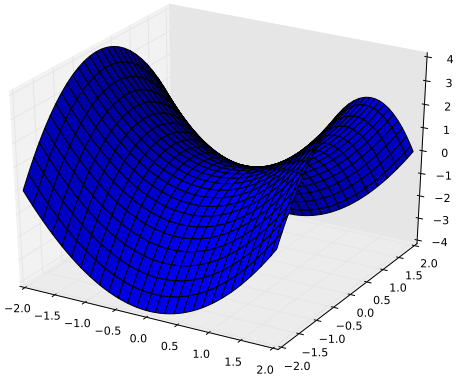
Ce module **numpy.random** possède aussi des fonctions simulant les lois de probabilité classiques : **rd.binomial(n,p)**, **rd.poisson(p)** etc.). Ce sera à approfondir en TSI2.

9. Le module mpl_toolkits.mplot3d

Le module **mpl_toolkits.mplot3d** permet des tracés en dimension 3.
La fonction la plus utile est **Axes3D** et on l'importe avec **from mpl_toolkits.mplot3d import Axes3D**.
Remarquons que **mpl_toolkits.mplot3d** par sa syntaxe est un module de la bibliothèque **mpl_toolkits** mais c'est le seul module de cette bibliothèque qui peut nous intéresser à notre niveau.

Exemple 09. *Tracer la surface* $z = x^2 - y^2$ *pour* $(x, y) \in [-2, 2]^2$ *avec un pas* $h = 0.015$.

```
In[23]: import numpy as np; import matplotlib.pyplot as plt
In[24]: from mpl_toolkits.mplot3d import Axes3D
In[25]: def f(x,y) : return x**2 -y**2
In[26]: X = np.arange(-2,2,0.015); Y = np.arange(-2,2,0.015);
In[27]: X,Y = np.meshgrid(X,Y); ax = Axes3D(plt.figure())
In[28]: Z = f(X,Y); ax.plot_surface(X,Y,Z); plt.show( )
```



10. Le module time

Le module **time** gère tout ce qui concerne le temps d'exécution. Il permet de connaître le temps mis pour l'exécution d'une procédure donnée. Quand vous aurez vu les fonctions récursives, ce module permettra de comparer le temps mis pour faire un calcul de façon récursive plutôt qu'itérative (avec une boucle **for**). Ce module peut illustrer aussi la comparaison entre une procédure en $O(n)$ et celle en $O(n^2)$ (voir le prochain chapitre). La fonction utile de **time** est **perf_counter**. Le mieux c'est donner un exemple.

Exemple 10. *Soit la suite* $(F_n)_{n\in\mathbb{N}}$ *définie par* $F_0 = 1$, $F_1 = 1$, *et* $\forall n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$.
Écrivons une procédure **Fibo1** *qui à partir de l'entier* n *renvoie* F_n, *en utilisant directement la relation de récurrence et de façon itérative. Appliquons pour* $n \in [\![1, 14]\!]$.
Écrivons ensuite une procédure **Fibo2** *qui à partir de l'entier* n *renvoie* F_n, *en utilisant directement la relation de récurrence et de façon récursive. Appliquons pour* $n \in [\![1, 14]\!]$.
Ensuite, comparons le temps d'exécution dans les deux cas pour $n = 20$ *avec le module* **time**.

```
In[1]: def Fibo1(n) :
           if n == 0 or n ==  1 :
                 return 1
           else :
                 u , v  = 0 , 1
                 for j in range(n-1):
                 u , v = v , u+v
                 return v
# On fait fonctionner.
In[2]: [Fibo1(i) for i in range(1,15)]
Out[2]:
    [1,2,3,5,8,13,21,34,55,89,144,233,377,610]
```

```
In[3]: def Fibo2(n) :
           if n  <=  1 :
                 return 1
           else :
                 return(Fibo2(n-1)+Fibo2(n-2))
# On fait encore une fois fonctionner.
In[4]: [Fibo2(i)  for i in range(1,15)]
Out[4]
    [1,2,3,5,8,13,21,34,55,89,144,233,377,610]
```

Rien à dire, c'est pareil niveau résultat. Mais niveau temps d'exécution c'est pas pareil.
On commence à charger **perf_counter** avec **pc** pour alias. On visualise la valeur trouvée pour 20 avec **print** mais ce n'est pas obligatoire. Puis on tape **t = pc()** puis on tape le code d'execution ici **Fibo1(20)** pour le premier et on tape ensuite **print(pc()-t)** qui va afficher le temps mis pour l'execution.

```
In[5]: from time import perf_counter as pc
In[6]: print(Fibo1(20))
In[7]: t = pc( ); Fibo1(20); print(pc( ) - t)
Out[7]
      10946
      0.00014680592257036197

In[8]: print(Fibo2(20))
In[9]: t = pc( ); Fibo2(20); print(pc( ) - t)
Out[9]:
      10946
      0.0184203504243996
```

Fibo2(20) met 126 fois plus de temps que **Fibo1(20)**. Essayer avec 200 à la place de 20 et ce sera pire !

11. Le module decimal

Un dernier module pour la route. En fait, on vous donne ce module car il peut être pratique. Dans certains programmes, on peut avoir des soucis d'arrondis. Cela se traduira par un résultat obtenu faux s'il y a eu plusieurs étapes d'arrondis pour obtenir ce résultat. Pour contrer cela, on peut imposer plus de chiffres après la virgule. Donnons un exemple.

```
In [17]: import decimal

In [18]: D = decimal.Decimal

In [19]: decimal.getcontext().prec = 40

In [20]: print(1/7)
0.14285714285714285

In [21]: print(D(1)/D(7))
0.1428571428571428571428571428571428571429
```

Comme on a toujours voulu connaître plein de décimales de π ou de e (la constante de Neper), avanti !

```
In [26]: import math as mt
In [27]: print(mt.pi)
3.141592653589793
In [28]: print(mt.e)
2.718281828459045

In [29]: print(D(mt.pi))
3.1415926535897931159979634685441851615905761718 75

In [30]: print(D(mt.e))
2.7182818284590450907955982984276488423347473144531 25
```

Et maintenant, sur Internet, on va chercher les décimales de π jusqu'à 50 décimales par exemple.

$$\text{Pi} = 3.\ 1415926535\ 8979323846\ 2643383279\ 5028841971\ 6939937510$$

Et on voit qu'après la 15$^{\text{ème}}$ décimale, **`print(D(mt.pi))`** renvoie un résultat faux. Il faut savoir que le module **`math`** ne connaît les constantes usuelles dont π et e qu'avec 15 décimales. Et **`decimal`** retourne donc un résultat faux si l'on veut mieux.

Chapitre 6

Analysis of an algorithm

Objectifs

- **Les incontournables :**
  - savoir reconnaître et utiliser un type d'instruction ;
  - aborder l'effet de bord ;
  - percevoir l'utilité de commentaires pour mieux comprendre un bloc d'instructions ou un choix de programmation ;
  - savoir décrypter et gérer un message d'erreur ;
  - être capable de repérer des cas simples à tester ;
  - distinguer et traduire les cas limites dans un jeu de tests ;
  - connaître quelques ordres de grandeurs de complexité d'algorithmes classiques ;
  - distinguer et étudier les notions de complexité en temps et de complexité en espace.
- **Et plus si affinités :**

Cours

■ Introduction

Après avoir écrit des programmes et réfléchi au rôle de code, il est important d'étudier le contenu avec précision. On doit aussi analyser ce contenu et vérifier que le résultat obtenu lors de l'exécution est celui attendu. Il est primordial que le programme fonctionne correctement. Sinon, on doit le déboguer. Par ailleurs, il est intéressant de se pencher sur la complexité d'un algorithme ce qui permet de les classer niveau performance.

■ Les syntaxes d'une instruction

1. Deux instructions particulières : expression et affectation

Une **expression** est utilisée pour calculer une valeur ou pour appeler une procédure. Une expression est composée de noms, de nombres, de chaînes de caractères, d'éléments séparés par des signes de ponctuation, de crochets, de parenthèses, d'opérateurs (warning pas `=` qui n'est pas un opérateur) etc. Une expression a une valeur.

Une **affectation** est un autre type d'instruction et lie un nom à une valeur ou modifie un élément d'un objet mutable (comme une liste). Le symbole `=` est utlisé, rappelons-le, pour l'affectation.

2. Cas général

De façon générale, instruction est un morceau de code minimal qui produit un effet. Une instruction est exécutée par la machine. Attention, une instruction ne correspond pas toujours à une ligne de codes. On peut avoir plusieurs instructions séparées par des points virgules dans une ligne. On parle d'instruction **simple**. Ce sont en général des expressions ou des affectations.

On peut avoir aussi une instruction qui prend toute une ligne et qui affecte plusieurs lignes. On parle d'instruction **composée**.

Une instruction simple peut être une affectation, une fonction prédefini par exemple `assert` ou `return` ou `break` ou `import` etc.

Une instruction composée est une instruction sur une ligne terminée par deux points `:` suivie d'une ou plusieurs instructions indentées. On pense à une boucle conditionnelle du type `if` par exemple suivis de `elif` ou `else`. On peut penser aussi à une boucle `for` ou à la définition d'une fonction avec `def`.

Une instruction (hormis une expression) n'a pas de valeur. Certaines instructions nécessitent une expression comme le signe `=` suivis d'une valeur. Ces choix de conception du langage Python ont des conséquences à connaître. Le mieux c'est taper des exemples.

Exemple 01 Une manipulation de `assert`, fonction déjà entrevue dans le chapitre 1.

```
In [1]: x = 1
In [2]: assert x = 1
  File "<ipython-input-4-d93c3e0517ec>", line 1
    assert x=1
           ^
SyntaxError: invalid syntax
In [3]: assert x == 1
```

Explication : écrire **`assert x=1`** provoque une erreur de syntaxe même si l'on a tapé avant **`x=1`**. Par contre **`assert x==1`** est une bonne syntaxe, en fait il renvoie alors **`True`** sans l'afficher.

Exemple 02 Manipulation d'une expression suivant **`if`**.

```
In [10]: x =3
In [11]: if x%2 == 1:
    ...:     x=x+1
In [12]: x
Out[12]: 4
In [13]: x=5
In [14]: if x%2 :
    ...:     x=x+1
In [15]: x
Out[15]: 6
```

Explication : On remarque que l'instruction **`x%2 == 1`** et l'instruction **`x%2`** après **`if`** sont similaires. En effet, le reste de la division euclidienne par 2 donne 0 ou 1. Tout nombre nul est interprété comme **`False`** et la boucle **`if`** ne fonctionnera que si **`x%2`** vaut 1 et alors considérée comme vraie.

Il faut savoir que les mots **`if, elif, while, assert`** doivent être suivis d'une expression. Cette expression n'a pas forcément une valeur booléenne mais quelle que soit sa valeur, elle est interprétée comme telle. Ainsi par exemple :

```
In [16]: assert not print("!!!")
!!!

In [17]: assert print("!!!")
!!!
Traceback (most recent call last):

  File "<ipython-input-17-9f325eaebad1>", line 1, in <module>
    assert print("!!!")

AssertionError
```

Ainsi **`assert not print("!!!")`** ne provoque pas de message d'erreur.
En effet, **`print("!!!")`** est une expression de type **`None`**

```
In [18]: type(print("!!!"))
!!!
Out[18]: NoneType
```

Et l'expression `not None` a pour valeur `True`.
De façon générale, tout nombre nul et tout conteneur vide est interprété comme une valeur `False`. C'est le cas pour `0, 0.0, "␣", [], (), , None`. Toute autre valeur est interprétée comme `True`. C'est pourquoi par exemple `if x% 3` renvoie `True` pour deux valeurs 1 et 2.
Donc l'égalité `==` est ici obligatoire si l'on veut différencier ces deux cas.

■ Effet de bord ou side effect

On dit qu'une fonction a un **effet de bord ou side effect** si son exécution modifie quelque chose en dehors de ce qui est défini dans le corps de la fonction, par exemple un de ses paramètres ou une variable globale définie dans le programme. Pour que ça semble plus clair dans votre esprit, donnons deux exemples.

Exemple 03. Nous allons créer deux fonctions qui ajoute `x` à une liste `liste`.

```
In [2]: def ajoute01(liste,x):
   ...:     liste.append(x)

In [3]: def ajoute02(liste,x):
   ...:     liste = liste + [x]
   ...:     return liste

In [4]: liste = [1,2,3,4]
In [5]: ajoute01(liste,5)
In [6]: liste
Out[6]: [1, 2, 3, 4, 5]

In [7]: liste = [1,2,3,4]
In [8]: ajoute02(liste,5)
Out[8]: [1, 2, 3, 4, 5]
In [9]: liste
Out[9]: [1, 2, 3, 4]
```

On remarque qu'après l'action de `ajoute01`, `liste` a été modifié (ici 5 a été ajouté en fin de liste). C'est `append` qui modifie la liste initiale. Il y a donc un effet de bord pour `ajoute01`.
Par contre l'action de `ajoute02` ne modifie pas `liste` quand on l'appelle. En fait, dans `ajoute02`, on crée une variable locale `liste` qui n'est pas `liste` hors procédure. Et comme toute variable locale, elle est oubliée après l'exécution de la procédure.

Exemple 04. Donnons un exemple un peu plus subtil. Nous allons créer deux fonctions qui multiplie `x` aux éléments d'une liste `liste`.

```
In [10]: def multiplie01(liste,x):
    ...:     res = liste
    ...:     for i in range(len(res)):
    ...:         res[i] = x * res[i]
    ...:     return res

In [11]: def multiplie02(liste,x):
    ...:     res = liste
    ...:     return [x*res[i] for i in range(len(res))]
```

Dans **multiplie01**, **liste** qui n'est jamais une variable locale est récupérée par la variable locale **res** dont le contenu est modifié par la procédure. Et donc **liste** se retrouve modifiée.
Il y a effet de bord.
Par contre dans **multiplie02**, bien que **res** récupère aussi **liste**, cette variable **res** n'est pas modifiée car on retourne une nouvelle liste créée à partir de **res**.
Illustrons-le avec **liste**, où on a affecté **[1,2,3,4]** au départ.
On multiplie par 5 par les deux procèdures.

```
In [12]: multiplie01(liste,5)
Out[12]: [5, 10, 15, 20]

In [13]: liste
Out[13]: [5, 10, 15, 20]

In [14]: liste = [1,2,3,4]
In [15]: multiplie02(liste,5)
Out[15]: [5, 10, 15, 20]

In [16]: liste
Out[16]: [1, 2, 3, 4]
```

■ Specification d'une fonction et annotations

1. Spécification ou docstring
Une spécification permet d'informer les utilisateurs de la tâche effectuée par la fonction, de préciser les contraintes imposées pour les paramètres et ce qui peut être attendu des résultats. Elle peut aussi préciser les messages d'erreurs affichés en cas de mauvaise utilisation. Elle est résumée dans la **docstring** inscrite au début du corps de la fonction entre des triples guillemets.
On peut créer son docstring en créant la fonction ou la procédure. On le placera avant le corps de la procédure mais après la déclaration du nom de la fonction avec le modèle suivant :

```
def nom_de_la_procedure(arguments):
    """ corps du docstring : informations sur la fonction, non
        obligatoires mais recommandees """
    corps de la procedure
```

Attention, le docstring est destiné à l'utilisateur. Il n'est pas utilisé par l'interpréteur Python.
On peut récupérer le docstring d'une fonction ou d'une procédure avec **`help`**

Exemple 04 bis. Reprenons **`multiplie02`** en lui mettant un docstring.

```
In [1]: def multiplie02(liste,x):
   ...:     """ Pas d'effet de bord. Chouetteeeeeee !!!!! """
   ...:     res = liste
   ...:     return [x * res[i] for i in range(len(res))]
In [2]: help(multiplie02)
Help on function multiplie02 in module __main__:

multiplie02(liste, x)
    Pas d'effet de bord. Chouetteeeeeee !!!!!
```

Les fonctions prédéfinies ont aussi leurs docstrings.

Exemple 05. Partons à la découverte de la fonction **`divmod`**

```
In [3]: help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

Analysons ce qui a été affiché. La fonction **`help`** est allé chercher le docstring de **`divmod`**. La fonction a deux arguments **`x`** et **`y`** et renvoie un couple composé du quotient et du reste de la division euclidienne de **`x`** par **`y`**. L'égalité **`div*y+mod == x`** signifie que si l'on rentre dans **`div`** (respectivement **`mod`**) le premier élément (respectivement le second élément) du couple, **`x`** s'obtient en faisant **`div*y+mod`** ce qui est bien la division euclidienne de **`x`** par **`y`**.

```
In [4]: x = 7.2; y = 3
In [5]: div,mod = divmod(x,y)
In [6]: div
Out[6]:
       2.0

In [7]: mod
Out[7]:
       1.2000000000000002

In [8]: div*y + mod == x
Out[8]:
       True
```

On peut remarquer que **`x`** et **`y`** ne sont pas obligatoirement entiers pour faire la division euclidienne.

Exemple 06. Je désire connaître mieux **hypot** du module **math**

```
In [12]: import math

In [13]: help(math.hypot)

Help on built-in function hypot in module math:

hypot(...)
    hypot(*coordinates) -> value
    Multidimensional Euclidean distance from the origin to a
       point.
    Roughly equivalent to:
        sqrt(sum(x**2 for x in coordinates))
    For a two dimensional point (x, y), gives the hypotenuse
    using the Pythagorean theorem:  sqrt(x*x + y*y).
    For example, the hypotenuse of a 3/4/5 right triangle is:
       >>> hypot(3.0, 4.0)
       5.0
```

On remarque que le docstring est plus ou moins complet selon la fonction appelée.
Python est plus prolifique pour **math.hypot** que pour **divmod**.

2. Annotations et commentaires

Un programme doit être lu et relu facilement par l'auteur mais aussi par quelqu'un qui dćouvre le programme. Aux concours notamment les annotations et commentaires ont vraiment leur place. Si le candidat doit fabriquer sa procédure et qu'elle dépasse quelques lignes de codes, il faut mettre des annotations pour expliquer ce que fait telle ligne de code ou telle boucle. Souvent dans les sujets il y a des procédures à trous où le candidat doit rajouter les lignes ou les morceaux de ligne de code manquants. Dans ces procédures, il y a souvent des annotations dont le rôle est de guider pour remplir ce qui manque. Le docstring (voir le paragraphe précédent) joue aussi ce rôle.
Un commentaire est toujours précédé de **#** et comme pour un docstring, un commentaire n'est pas utilisé par l'interprétateur Python. Remarquons qu'un commentaire peut être seul sur une ligne ou se placer après du code sur la même ligne que ce code.

Exemple 07

Nous allons créer une procédure **Swap_first_last** d'argument **liste** qui permute le premier et le dernier élément d'une liste et mettons-y un docstring assez complet et des annotations pour expliquer les lignes de codes.

```
In [14]: def Swap_first_last(liste):
    ...:     """ liste est de type list
    ...:     la procedure permute le premier et le dernier
       element et renvoie une nouvelle liste
    ...:     Swap_first_last([1,2,3,4]) renvoie [4,2,3,1] """
    ...:     n = len(liste) # n est la longueur de liste
    ...:     # On fait une copie de liste :
    ...:     copie = liste[:]
    ...:     # on permute les deux elements extremes de liste :
    ...:     copie[0], copie[n-1] = copie[n-1], copie[0]
    ...:     return copie
```

```
In [15]: liste = [-5,8,7,1,2,4]

In [16]: Swap_first_last(liste)
Out[16]:
        [4, 8, 7, 1, 2, -5]

In [17]: liste
Out[17]:
        [-5, 8, 7, 1, 2, 4]
```

On peut remarquer en passant que **`Swap_first_last`** n'a pas d'effet de bord.

■ Assertion

Pour l'instant, on s'est intéressé dans une procédure à comment l'expliquer, faire comprendre ce qu'elle fait, quelles variables on utilise etc. On va maintenant rajouter des instructions qui vont arrêter le programme en cas de mauvaise utilisation.

On utilise une **assertion** qui est l'affirmation qu'une proposition est vraie. La fonction clé qui va être utile, vous l'avez peut-être deviné, c'est la fonction **`assert`** que l'on commence à connaître. On rappelle que l'on utilise **`assert`** suivi d'une expression dont la valeur est interprétée comme une valeur booléenne. Comme on a vu plus haut, si l'expression a la valeur **`True`**, il ne se passe rien. Sinon, le programme est interrompu et un message d'erreur s'affiche **`AssertionError`**

Exemple 08. Etablissons un programme qui renvoie l'inverse d'un flottant. L'idée c'est qu'il faut signaler quand un très mauvais élève veut trouver l'inverse de 0.

```
In [19]: def inverse(x):
    ...:     """ x est un nombre non nul de type int ou float et
       cette fonction renvoie l'inverse de x """
    ...:     assert x!= 0
    ...:     return 1/x

In [20]: inverse(-3)
Out[20]:
        -0.3333333333333333

In [21]: inverse(0)
Traceback (most recent call last):

  File "<ipython-input-21-7538d73c586c>", line 1, in <module>
    inverse(0)

  File "<ipython-input-19-49acf1e29ca6>", line 3, in inverse
    assert x!= 0

AssertionError
```

Que se passe t-il si l'on ne borde pas le cas de la division par 0.

```
In [22]: def new_inverse(x):
    ...:     return 1/x

In [23]: new_inverse(0)

Traceback (most recent call last):

  File "<ipython-input-23-a310b552216b>", line 1, in <module>
    new_inverse(0)

  File "<ipython-input-22-0ed19f52f469>", line 2, in new_inverse
    return 1/x

ZeroDivisionError: division by zero
```

Finalement, pas grand chose de plus ! Et ici l'erreur commise est bien expliquée.

Exemple 09. *Reprendre la fonction* **`maximum`** *créée au chapitre 3 page 39 en y rajoutant un docstring disant que L doit etre non vide et ce qu'elle renvoie et en y rajoutant aussi un assert qui interrompt le programme si L est vide. On nommera* **`new_maximum`** *cette fonction.*

```
In [6]: def new_maximum(L):
   ...:       """ L est une liste de nombres non vide qui renvoie
       le maximum de la liste L """
   ...:       assert L != []
   ...:       maxi = L[0]
   ...:       for i in range(1,len(L)):
   ...:           if L[i] > maxi :
   ...:               maxi = L[i]
   ...:       return maxi

In [7]: new_maximum([1,7,0,-8,7,10,2])
Out[7]:
        10
In [8]: new_maximum([])

Traceback (most recent call last):

  File "<ipython-input-8-c6997c0edb32>", line 1, in <module>
    new_maximum([])

  File "<ipython-input-6-7a1839672011>", line 3, in new_maximum
    assert L != []

AssertionError

# Et juste pour le fun :

In [10]: help(new_maximum)
Help on function new_maximum in module __main__:

new_maximum(L)
    L est une liste de nombres non vide qui renvoie le maximum de
        la liste L
```

Donnons un dernier exemple dans le monde des dictionnaires :

```
In [11]: def fonction(dico,k):
              """ dico est un dictionnaire
              k est une cle de dico """
              assert k in dico
              corps de la procedure
```

On contrôle ici si la valeur passée en second argument est bien une clé du dictionnaire mis en premier argument. Si tel n'est pas le cas, le programme s'interrompt et **`Assertion Error`** s'affiche.

■ Tests d'un programme

1.Introduction

Même si l'on soit clair dans le docstring, que l'on mette des commentaires aux endroits stratégiques du programme et que l'on borde aussi avec des**assert** pour éviter de rentrer n'importe quoi en argument, il peut se passer des choses étranges dans l'exécution du programme qui peut très bien ne pas nous sauter aux yeux de suite. Il y a plusieurs sortes de bugs.

- Une erreur survient lors de certains tests et pas avec d'autres, c'est un bug intermittent. Ce cas est un peu vicieux comme on va le voir avec l'exemple en dessous;
- une erreur survient pour chaque test, c'est un bug permanent; on a certainement tapé un mauvais code quelque part, par exemple on a écrit une variable locale avec une faute d'orthographe, mettre un **=** alors que c'est un **==**, etc.;
- le programme s'arrête de manière prématurée ou ne s'arrête pas, le bug est visible. Là c'est soit encore une erreur de frappe, soit un souci dans la conception de l'algorithme.

Exemple 10. Nous allons construire deux fonctions, la première **Strange01(x,y)** retourne $\sqrt{xy}$ et la seconde **Strange02(x,y)** retourne $\sqrt{x}.\sqrt{y}$. Nous utiliserons la puissance 0.5 pour la racine carrée. On peut utiliser aussi **sqrt** de **numpy** mais quand c'est possible, c'est mieux sans package.

```
In [11]: def Strange01(x,y):
    ...:     return (x*y) ** 0.5
In [12]: def Strange02(x,y):
    ...:     return (x ** 0.5) * (y ** 0.5)
In [13]: Strange02(1e152,1e152)/Strange01(1e152,1e152)
Out[13]: 1.0
In [14]: Strange02(1e155,1e155)/Strange01(1e155,1e155)
Out[14]: 0.0
# Juste pour la remarque de l'enonce :
In [15]: import numpy as np
In [16]: np.sqrt(9)
Out[16]: 3.0
```

Comme x et y sont positifs, $\sqrt{xy} = \sqrt{x}\sqrt{y}$ et normalement les deux fonctions donnent le même résultat. Et pourtant, c'est le cas pour $x = y = 10^{152}$ et ce n'est pas le cas pour $x = y = 10^{155}$. Est-ce un problème d'arrondi. Voyons avec le module **decimal** vu en fin de chapitre 5.

```
In [17]: import decimal
In [18]: D = decimal.Decimal
In [19]: decimal.getcontext().prec = 40
In [20]: D(Strange02(1e155,1e155))/D(Strange01(1e155,1e155))
Out[20]: Decimal('0E-1000038')
# Ce n'est pas bon. Augmentons la precision
In [21]: decimal.getcontext().prec = 800
In [22]:  D(Strange02(1e155,1e155))/D(Strange01(1e155,1e155))
Out[22]: Decimal('0E-1000798')
```

Donc le problème reste compliqué. Bref, évitons les nombres très, très grands (ou petits) !

2. Construction d'un jeu de tests

Construire un jeu de tests consiste à définir un ensemble de données qui vont être utilisées pour vérifier que le programme produit bien les résultats attendus acec ces données. Ces vérifications peuvent se faire de plusieurs manières. On peut utiliser **`print`** pour afficher quelques réponses et vérifier qu'elles sont correctes. Pour un nombre conséquent de tests, il est pratique d'utiliser aussi des assertions. Un message d'erreur est affiché seulement pour les cas qui posent problème.

On distinguera plusieurs types de tests :

- Tester quelques cas simples typiques (pour une utilisation basique du programme);
- tester des valeurs extrêmes, des cas limites, des cas interdits;
- tester un nombre important de données (choisies de façon aléatoire par exemple);
- tester des cas qui pourraient nécessiter un temps d'exécution important afin d'évaluer l'efficacité du programme.

Une idée pour effectuer ces tests, c'est de partitionner le domaine d'entrée. Par exemple si l'on désire calculer le PGCD de deux entiers m et n, c'est-à-dire le plus grand diviseur commun (ainsi 5 est le PGCD de 15 et de 10), on fait un programme personnel **`LEPGCD`** d'arguments **`m`** et **`n`** qui retourne le PGCD de **`m`** et de **`n`**. Puis on peut tester avec un cas où **`m > n`** puis un autre avec **`m`**=**`n`** puis un cas avec **`m`** est un multiple de **`n`** etc.
Si l'on a créée une fonction avec une liste à l'entrée, on teste naturellement le cas d'une liste vide et le cas d'une liste à un élément ou le cas d'une liste avec un élément qui se répete un certain nombre de fois, etc.

Exemple 07 bis. Reprenons la fonction **`Swap_first_last`** qui inverse le premier et dernier élément d'une liste.

```
In [14]: def Swap_first_last(liste):
    ...:     """ liste est de type list
    ...:     la procedure permute le premier et le dernier
             element et renvoie une nouvelle liste
    ...:     Swap_first_last([1,2,3,4]) renvoie [4,2,3,1] """
    ...:     n = len(liste) # n est la longueur de liste
    ...:     # On fait une copie de liste :
    ...:     copie = liste[:]
    ...:     # on permute les deux elements extremes de liste :
    ...:     copie[0], copie[n-1] = copie[n-1], copie[0]
    ...:     return copie
```

On essaye des cas différents. Si **`assert`** ne renvoie rien, c'est que l'égalité est **`True`**.

```
In [2]: assert Swap_first_last([1,2,3,4]) == [4,2,3,1]

In [3]: assert Swap_first_last([[1,2],['oui','non'],['OK','KO']])
        == [['OK','KO'],['oui','non'],[1,2]]

In [4]: assert Swap_first_last([1]) == [1]
```

Par contre, si l'on tape :

```
In [5]: assert Swap_first_last([]) == []
Traceback (most recent call last):

  File "<ipython-input-5-8e4cebbbed72>", line 1, in <module>
    assert Swap_first_last([]) == []

  File "<ipython-input-1-c90e4594a822>", line 9, in
     Swap_first_last
    copie[0], copie[n-1] = copie[n-1], copie[0]

IndexError: list index out of range
```

La fonction **`Swap_first_last`** ne traite pas le cas d'une liste vide. Changeons le programme.

```
In [6]:  def New_Swap_first_last(liste):
    ...:      """ liste est de type list
    ...:      la procedure permute le premier et le dernier
              element et renvoie une nouvelle liste
    ...:      Swap_first_last([1,2,3,4]) renvoie [4,2,3,1] """
   ...:       if liste == [] : return []
    ...:      n = len(liste) # n est la longueur de liste
    ...:      # On fait une copie de liste :
    ...:      copie = liste[:]
    ...:      # on permute les deux elements extremes de liste :
    ...:      copie[0], copie[n-1] = copie[n-1], copie[0]
    ...:      return copie

In [7]:  New_Swap_first_last([])
Out[7]: []
```

Cela marche !

Exemple 11. Une fonction prend en argument deux listes de même longueur. La deuxième liste est modifiée et contient à la fin les mêmes éléments que la premirer liste aux mêmes places s'ils sont positifs ou nuls et des zéros sinon. Nous allons faire deux versions et nous rendre compte qu'une des deux ne marche pas toujours.
Dans la première version **`ESSAI01`**, on crée une boucle dans laquelle on remplace les éléments de **`liste2`** de l'indice 0 à **`len(liste1)-1`** par ceux de **`liste1`** de même indice si cet élément de **`liste1`** est positif. Sinon on y met 0. On retourne enfin **`liste2`** transformé.

```
In [1]: def ESSAI01(liste1,liste2):
   ...:     for i in range(len(liste1)):
   ...:         if liste1[i] >= 0 :
   ...:             liste2[i] = liste1[i]
   ...:         else:
   ...:             liste2[i] = 0
   ...:     return liste2
```

Dans la seconde procédure **ESSAI02**, on commence par remplacer tous les éléments de **liste2** par des 0. Puis on remplace ces 0 par les éléments de **liste1** correspondants à leur indice si ces éléments de **liste1** sont positifs.

```
In [8]: def ESSAI02(liste1,liste2):
   ...:     for i in range(len(liste2)):
   ...:         liste2[i] = 0
   ...:     for i in range(len(liste2)):
   ...:         if liste1[i] >= 0 :
   ...:             liste2[i] = liste1[i]
   ...:     return liste2
```

Faisons des essais.

```
In [10]: liste1 = [2,-3,5,-1]

In [11]: liste2 = [1,2,3,4]

In [12]: ESSAI01(liste1,liste2)
Out[12]: [2, 0, 5, 0]

In [13]: ESSAI02(liste1,liste2)
Out[13]: [2, 0, 5, 0]
```

Pour l'instant tout est OK.

```
In [16]: ESSAI01(liste1,liste1)
Out[16]: [2, 0, 5, 0]

In [17]: ESSAI02(liste1,liste1)
Out[17]: [0, 0, 0, 0]
```

L'explication c'est que **liste1** est devenu **liste2** et au bout de la première boucle de **ESSAI02**, **liste1** ne contient plus que des 0. Puis on reremplit **liste2** des 0 de **liste1** dans la deuxième boucle car **liste1[i] >= 0** est toujours vraie.

■ Complexité d'un algorithme

1. Introduction

Un algortithme, en dehors du fait qu'il doit être correct, doit satisfaire à des deux impératifs en terme de consommation de ressources :

- Utiliser un espace en mémoire acceptable, on parle de **complexité en espace**;
- produire la réponse attendue en un temps acceptable, on parle de **complexité temporelle**.

Le temps d'exécution dépend de la machine, du langage de programmation, de l'algorithme.
On ne peut pas jouer généralement sur les deux premiers mais beaucoup plus sur le troisième.
C'est sa complexité temporelle qu'il faut analyser.

Nous supposerons que le temps d'exécution d'une affectation, d'une opération arithmétique simple, d'une comparaison sont pratiquement identiques.
Ces temps d'exécutions constituent une unité de base.
Ainsi une permutation de deux éléments correspond à une double affectation donc vaut deux affectations.

Nous posons les règles suivantes :

- Le temps d'exécution d'une suite d'instructions est la somme des temps d'execution de chaque instruction.
- Le temps d'exécution d'une instruction conditionnelle du type `if text : instructions 1` suivi de `else : instructions 2` est inférieur ou égal au maximum du temps d'exécution de `instructions 1` et `instructions 2` plus le temps d'exécution du test.
- Le temps d'exécution d'une boucle `for i in range(p) : instructions` est infop fois le temps d'exécution de `instructions` si ce temps est constant plus `p` (le nombre d'affectations pour la variable `i`. Si le temps d'exécution de `instruction` dépend de la valeur de `p`, l'étude se ramène au cas par cas.

  Enfin, pour une boucle `while` l'étude se ramène aussi au cas par cas.

L'évaluation du temps d'execution d'un algorithme se réduit ainsi à une évaluation en fonction d'un nombre n (entier représentant la taille des données en entrée), du nombre total d'opérations élémentaires noté u_n. Le niveau de complexité correspond au type de croissance de la suite (u_n).

Suivant les valeurs de l'entrée, u_n peut prendre des valeurs très différentes. Si, par exemple, nous parcourons une liste à l'aide d'une boucle, à la recherche d'un élément, celui-ci peut se trouver en premier et nous sortons de la boucle, c'est **le cas le plus favorable**.
Il peut se trouver à la fin de la liste, c'est **le pire des cas**.

2. Niveaux de complexité

Considérons deux suites (u_n) et (v_n). On suppose qu'aucune des deux suites ne s'annulent à partir d'un certain rang de n.

- **Notion de grand O**

Si le rapport $\left|\dfrac{u_n}{v_n}\right|$ est borné à partir d'un certain rang de n, alors $u_n = O(v_n)$.

(On dit que u_n est un grand O de v_n quand n tend vers $+\infty$). On peut l'écrire :

$$\exists\, K \in \mathbb{R}^+,\ \exists\, N \in \mathbb{N},\ \forall n \geqslant N,\ |u_n| \leqslant K|u_n|.$$

Par exemple, si l'on pose $u_n = n + 2024\ln n$ et $v_n = 2n$, $\left|\dfrac{u_n}{v_n}\right| = \left|\dfrac{n + 2024\ln n}{2n}\right| = \dfrac{1}{2} + \dfrac{2024\ln n}{2n}$.

Comme $\lim\limits_{n\to+\infty} \dfrac{\ln n}{n} = 0$, il existe $N \in \mathbb{N}$ tel que $\dfrac{2024\ln n}{2n} < 1$ pour $n \geqslant N$.

$$\forall n \geqslant N,\ \left|\frac{u_n}{v_n}\right| \leqslant \frac{3}{2} \Rightarrow u_n = O(v_n).$$

On peut remarquer que le cas le plus courant pour avoir $u_n = O(v_n)$ est de montrer que :

$$\lim_{n\to+\infty} \frac{u_n}{v_n} = l,$$

où l est un réel fini.

Un cas particulier important est $\lim\limits_{n\to+\infty} \dfrac{u_n}{v_n} = 0$, on écrit alors $u_n = o(v_n)$ et on dit que u_n est un petit o de v_n. Par exemple on a : $n = o(n^2)$.

On remarquera que $u_n = o(v_n) \Rightarrow u_n = O(v_n)$ mais on n'a pas la réciproque.

Par exemple si $u_n = n^2$ et $v_n = n^2 + n$, il est clair que $\lim\limits_{n\to+\infty} \dfrac{u_n}{v_n} = 1$ et donc $n^2 = O(n^2 + n)$ mais par contre : $n^2 + n \neq o(n^2 + n)$.

- **Complexité constante**

Si pour $n \to +\infty$, $u_n = O(1)$, cela signifie que le temps d'exécution est borné (indépendant de n). Par exemple, c'est le cas pour obtenir le premier élément d'une liste.

- **Complexité logarithmique**

Si pour $n \to +\infty$, $u_n = O(\ln n)$, on peut aussi écrire de façon équivalente, $u_n = O(\ln_2 n)$ ou $u_n = O(\ln_{10} n)$. On rappelle que $\ln_p n = \dfrac{\ln n}{\ln p}$. On double le temps d'exécution en élevant au carré la taille des données. C'est le cas avec la recherche dichotomique. On prend un intervalle où se situe l'élément cherché puis on coupe l'intervalle en deux puis encore en deux etc. en prenant soin que l'élément cherché est toujours dedans. La dichotomie est développée dans le cours plus tard dans l'année.

- **Complexité linéaire**

Si pour $n \to +\infty$, $u_n = O(n)$. Par exemple, on a une complexité linéaire pour le calcul de la somme de n termes avec une boucle non conditionnelle.

- **Complexité log-linéaire ou linéarithmique**

Si pour $n \to +\infty$, $u_n = O(n\ln n)$. C'est la complexité du meilleur tri par comparaison qu'il soit, c'est-à-dire le tri fusion. Ce sera vu en fin d'année ou en TSI2.

- **Complexité quadratique**

Si pour $n \to +\infty$, $u_n = O(n^2)$. Beaucoup d'algorithmes classiques ont cette complexité : le tri-bulle par exemple vu au chapitre 4. Généralement deux boucles imbriquées engendrent une telle complexité.

- **Complexité exponentielle**

Si pour $n \to +\infty$, $u_n = O(2^n)$ ou $u_n = O(k^n)$ avec $k > 1$. Ces algorithmes sont en pratique inutilisables pour de grandes valeurs de n.

3. Quelques exemples typiques

On a dit que généralement deux boucles imbriquées engendrent une complexité quadratique. Essayons de préciser.

```
In [3]: def Programmons01(n,k):
   ...:     a = 0
   ...:     for i in range(n) :
   ...:         for j in range(k):
   ...:             a += i*j
   ...:     return a
```

Par exemple `Programmons01(5,3)` retourne `30`. On suppose ici que k est fixé. Il y a une opération d'affectation au départ. Puis il y a n passages dans la boucle externe. À chaque passage dans cette boucle, on passe dans la boucle interne qui fait k multiplications puis k additions et k affectations. Au total, on a $u_n = 1 + 3kn$ opérations. On a alors $u_n = O(n)$.
Par rapport à n, la complexité est linéaire.

```
In [5]: def Programmons02(n):
   ...:     a = 0
   ...:     for i in range(n) :
   ...:         for j in range(n):
   ...:             a += i*j
   ...:     return a
```

Par exemple `Programmons02(5)` retourne `100`. Il y a une opération d'affectation au départ. Puis il y a n passages dans la boucle externe. À chaque passage dans cette boucle, on passe dans la boucle interne qui fait n multiplications puis n additions et n affectations.
Au total, on a $u_n = 1 + n.3n = 1 + 3n^2$ opérations.
On a alors $u_n = O(n^2)$. Par rapport à n, la complexité est quadratique.

```
In [8]: def Programmons03(n):
   ...:     a = 0
   ...:     for i in range(n) :
   ...:         for j in range(i):
   ...:             a += i*j
   ...:     return a
```

Par exemple `Programmons03(5)` retourne `35`. Il y a une opération d'affectation au départ. Puis il y a n passages dans la boucle externe. À chaque passage dans cette boucle, on passe dans la boucle interne qui fait i multiplications puis i additions et i affectations. Au total, on a :

$$u_n = 1 + \sum_{i=1}^{n} 3i = ++ \frac{3}{2} n(n+1)$$

opérations. On a encore $u_n = O(n^2)$. Par rapport à n, la complexité est encore quadratique.

Exercice 01. On admet que le nombre de chiffres dans l'écriture décimale de n est $\lfloor \log_{10}(n)+1 \rfloor$. On considère le programme suivant :

```
In [1]:  c = 0
In [2]:  while n > 0 :
    ...:      c = c + 1
    ...:      n = n // 10
```

1. Faire fonctionner ce programme à la main pour $n = 10008$. Combien y-a-t-il de tours dans la boucle ?
2. Déterminer la complexité par rapport à n.

Exercice 02.

1. Écrire un programme appelé **`Somme01(t)`** effectuant simplement avec deux boucles imbriquées le calcul de la somme

$$S = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} (t[i] - t[j])^2 \right)$$

pour un tableau t de nombres de longueur n.
Quelle est la complexité de cet algorithme en fonction de n ?

2. On pose pour tout $k \in [\![1, n]\!]$, la somme $S_k = \sum_{p=0}^{k-1} t[p]$ et pour tout $n \geqslant 2$, $W_n = \sum_{j=1}^{n-1} t[j] S_j$.
On admet que :

$$(1) : \quad S = 2\left((n-1)S_n^2 - 2nW_n\right).$$

   (a) On suppose que $n = 5$. Développer W_5 en fonction de $t[0], ..., t[4]$.

   (b) On considère le code :

```
In [1]: wn = 0 ; sn = 0
In [2]: for j in range(n-1):
            sn = sn + t[j]
            wn = wn + t[j+1]*sn
```

   Vérifier que pour $n = 5$ la valeur de **`wn`** obtenue est bien W_5.

   (c) En déduire le code de la fonction **`Somme02(t)`** qui retourne S calculé à partir de la formule (1).

   (d) Quelle est la complexité de cet algorithme en fonction de n ?

Chapitre 7

Sorting recursionless

Objectifs

- **Les incontournables :**
  - savoir

Cours

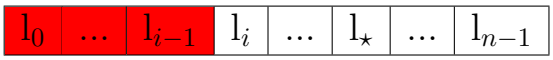
Le tri de données constitue l'un des problèmes fondamentaux de l'algorithmique. On considère une liste de données, auxquelles sont attachées des clés. On cherche à trier ces donnéesen fonction des clés (par exemple, trier une liste d'élèves par ordre alphabétique, ou par ordre de moyennes...). Nous nous placerons pour simplifier dans le cas ù la clé est la donnée elle-même.

■ Tri par selection

Principe du tri par sélection

Considérons une liste l de longueur n. Supposons que la plage $l[0..i-1]$
− soit triée
− contienne les i plus petits éléments de la liste

On initialise avec $i = 0$ ce qui permet de se calquer sur l'indéxation par défaut des listes pour Python, $l[0]$ est le premier élément de la liste l.
On selectionne, dans la sous-liste $l[i..n-1]$ le plus petit élément, que l'on permute avec l_i.
Sur le schéma plus bas, la partie en rouge est déjà triée et on permute alors l_i et $l_\star$ qui est le plus petit élément de $l[i..n-1]$.



Programmation

Commençons par écrire une fonction qui détermine l'indice du plus petit élément d'une liste entre les positions i et j.

```
In [1]: def cherche_ind_min(l,i,j):
   ...:     """renvoie l'indice du plus petit element de la sous-
       liste l[i..j]"""
   ...:     imin = i
   ...:     for k in range(i+1,j+1):
   ...:         if l[k] < l[imin]:
   ...:             imin = k
   ...:     return imin
In [2]: cherche_ind_min([-3,7,11,0,4,7,0,5,8,1],1,4)
Out[2]: 3
In [3]: cherche_ind_min([-3,7,11,0,4,7,0,5,8,1],0,9)
Out[3]: 0
```

Nous pouvons écrire alors la fonction de tri.

```
In [5]: def tri_selection(l):
   ...:     """trie la liste l par l'algorithme du tri selection,
       la fonction modifie la liste l"""
   ...:     n = len(l)
   ...:     for i in range(n-1):
   ...:         imin = cherche_ind_min(l,i,n-1)
   ...:         l[i],l[imin] = l[imin],l[i]
   ...:     return l
In [6]: tri_selection([-3,7,11,0,4,7,0,5,8,1])
Out[6]: [-3, 0, 0, 1, 4, 5, 7, 7, 8, 11]
```

Rappelons que, lors de l'écriture d'un algorithme, on doit être en mesure de justifier sa terminaison. C'est ici évident car on a une boucle inconditionnelle (boucle *for*) qui se termine toujours (contrairement à une boucle *while*).

Remarque : Lorsque les données à trier sont distinctes des clés utilisées pour trier la liste (par exemple, un élève n'est pas égal à sa moyenne), on peut s'intéresser à ce qui arrive à deux éléments ayant la même clé (deux élèves ayant la même moyenne) : après le tri, ces deux éléments sont-ils dans la même position relative l'un par rapport à l'autre qu'avant le tri ? Lorsque c'est le cas, on dit que le tri est **stable**. Le tri par sélection n'est pas stable : par exemple si les données sont A, B et C, et que les clés attachées sont 1,,1 et 0, (ce que l'on notera par A_1, B_1 et C_0), le tri par sélection part de $[A_1, B_1, C_0]$ qui devient $[C0, B1, A1]$ car le minimum de $(A1, B1, C0)$ est $C0$ qui permute avec $A1$.

Complexité

Calculons la complexité de cet algorithme en nous intéressant au nombre de comparaisons effectuées.

– la fonction *cherche_ind_min* appelée avec les arguments i et j, effectue $j - i$ comparaisons. Quand elle est appelée entre i et $n - 1$, elle en effectue $n - i - 1$.
– chaque passage dans la boucle indexée par i de la fonction *tri_selection* effectue donc $n - i - 1$ comparaisons. Comme i varie de 0 à $n - 2$, il y a au total :

$$C(n) = \sum_{i=0}^{n-2}(n - i - 1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \text{ comparaisons.}$$

La complexité vérifie $C(n) \in O(n^2)$. On parle de **complexité quadratique**.
Ce tri est donc peu efficace. Si la taille est multipliée par deux, le temps d'execution est multiplié par 4.

Exercice 01. *Trier A, B, C, D, E, F, G de clès respectives $1, 3, 8, 1, -2, 3, 5$* ***à la main*** *par le tri selection. On pourra noter la liste $[A_1, B_3, C_8, D_1, E_{-2}, F_3, G_5]$*

Exercice 02. ***Application : Utilisation du tri selection pour trier des points du plan selon leur distance à l'origine O du repère***

On dispose de points dans le plan muni d'un repère orthonormé d'origine O. Ces points possèdent un couple de coordonnées (x, y) représenté par la liste $[x, y]$. nous allons trier ces points en fonction de leur distance à 0, de la plus petite à la plus grande.

1. *Écrire une fonction* **`distance2`** *qui prend en paramètre une liste de deux nombres nommée* **`point`** *qui représente un point du plan, (*`point` *est la liste des coordonnées d'un point* P*), et renvoie le carré de la distance de ce point à* 0*. On pourra utiliser* **`sqrt`** *de* **`numpy`** *et le code* **`x,y=point`**.
Voyons cet exemple :

```
In [14]: x,y = [4,5]

In [15]: x
Out[15]: 4

In [16]: y
Out[16]: 5
```

2. *Écrire une fonction* **`compare`** *qui prend en paramètres deux mistes* **`p1`** *et* **`p2`** *représentant deux points* P_1 *et* P_2 *et qui renvoie* -1 *si* P_1 *est plus proche de* 0 *que* P_1 *et* 1 *si* P_2 *est plus proche de* 0 *que* P_1,, *et* 0 *si les deux points sont équidistants de* 0.
3. *On écrit une fonction* **`tri_points`** *qui prend en paramètre une liste composée de listes de deux nombres représantant des points du plan et qui trie cette liste suivant la distance entre les points et* O*. On adapte le tri selection. Le code est le suivant :*

```
In [1]:  def tri_points(liste):
    ...:      for i in range(len(liste)-1):
    ...:          i_mini = i
    ...:          mini = liste[i]
    ...:          for j in range(i+1, len(liste)):
    ...:              if compare(liste[j],mini) == -1:
    ...:                  i_mini = j
    ...:                  mini =liste[j]
    ...:          liste[i],liste[i_mini] = liste[i_mini],liste[i]
    ...:          return liste
```

Appliquer à la main cette fonction **`tri_points`** *à la liste* **`[[3,5],[2,1],[0,7],[-2,0]]`**
Solution.
1.

```
In [12]:  def distance2(point):
   ...:      x , y = point
   ...:      return np.sqrt(x*x + y*y)

In [13]: distance2([4,-1])
Out[13]: 4.123105625617661
```

2.

```
In [17]: def compare(p1,p2):
    ...:     d1 = distance2(p1)
    ...:     d2 = distance2(p2)
    ...:     if d1<d2:
    ...:         return -1
    ...:     elif d2<d2
    ...:         return 1
    ...:     else:
    ...:         return 0
In [18]: compare([7,8],[2,-7])
Out[18]: 1
In [19]: compare([-7,1],[2,-7])
Out[19]: -1
```

3.

```
In [24]: compare([0,7],[3,5]),compare([3,5],[2,1]),
Out[24]: (1,1)
In [25]: compare([2,1],[-2,0]), compare([0,7],[-2,0])
Out[25]: (1, 1)
In [26]: tri_points([[3,5],[2,1],[0,7],[-2,0]])
Out[26]: [[-2, 0], [2, 1], [0, 7], [3, 5]]
```

■ Tri par insertion

Principe du tri par insertion

C'est le tri du joueur de cartes qui reçoit ses cartes une à une. Le premier élément constitue une liste triée à lui tout seul. On compare ensuite le deuxième élément au premier pour savoir si on l'insère avant ou après le premier. À la $k^{\text{ème}}$ itération, on insère le $k+1^{\text{ème}}$ élément l_k dans la plage, déjà triée, des k premiers éléments, identifiée en rouge et ainsi de suite.

Programmation

```
In [1]: def tri_insertion(l):
   ...:     """trie la liste l par algorithme du tri selection,
      la fonction modifie la liste l"""
   ...:     n = len(l)
   ...:     for k in range(1,n):
   ...:         a_placer = l[k]
   ...:         i = k-1
   ...:         while i >= 0 and a_placer < l[i]:
   ...:             l[i+1] = l[i]
   ...:             i = i-1
   ...:         l[i+1] = a_placer
   ...:     return l
```

```
In [2]: tri_insertion([-1,-5,4,-7,1,10,8,0])
Out[2]: [-7, -5, -1, 0, 1, 4, 8, 10]
```

Analysons la terminaison du programme :
– la boucle *for* se termine ;
– la boucle *while* aussi car même si l'on ne trouve aucun indice i tel que l_i soit strictement supérieur à l'élément à placer, i diminue strictement à chaque étape donc finira par devenir négatif, ce qui terminera la boucle.
Remarquons d'ailleurs que l'ordre dans lequel les conditions sont écrites n'est pas anodin : on vérifie d'abord que $i \geqslant 0$ puis onlit l'élément dans la case d'indice -1, ce qui provoquerait une erreur).

Remarque : A la différence du tri par selection, le tri par insertion est stable : un élément de clé c ne peut passer devant un autre élément que si cet autre élément est de clé $c' > c$. Deux éléments de même clé ne sont jamais permutés.

Complexité

À la différence du tri par sélection, la complexité n'est pas la même pour toutes les listes de taille n. Nous allons étudier la complexité $C_{min}(n)$ dans le **meilleur des cas** et la complexité $C_{max}(n)$ dans le **pire des cas** (toujours en nous intéressant au nombre de comparaisons effectués entre éléments du tableau).
Dans le meilleur des cas, chaque élément l_k $(k \geqslant 1)$ est comparé à son prédécesseur et reste à sa place car $l_k \geqslant l_{k-1}$. Ce cas se produit lorsque le tableau est déjà trié ; la complexité est alors

$$C_{min}(n) = n - 1.$$

(On parle de complexité linéaire).
Dans le pire des cas, chaque élément l_k $(k \geqslant 1)$ est comparé à ses prédécesseurs et est finalement inséré en tête de liste. Ce cas se produit lorsque la liste est intialement triée à l'envers. La complexité est alors :

$$C_{max}(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}.$$

C'est encore une complexité quadratique (comme pour le tri par sélection).

Exercice 02. *Trier A, B, C, D de clès respectives $1, 3, 8, 1$* ***à la main*** *par le tri insertion. On détaillera les différentes affectations et on pourra noter la liste $[A_1, B_3, C_8, D_1]$.*

Exemple. Nous allons comparer le temps d'exécution du tri par insertion, par selection et par l'attribut prédéfini **`sort`** en utilisant le module **`time`**.

```
In [3]: from time import perf_counter as pc
In [4]: l = [2,1,4,0,8,4,-4,7,5,8,7,10,0,-4,-5,-8,4,11,7,9]
In [5]: t = pc( ); print(tri_selection(l)) ; print(pc( ) - t)

[-8,-5,-4,-4,0, 0,1,2, 4,4,4, 5,7,7,7,8,8,9,10,11]
0.0007508000001053006

In [6]: new_l =  [2,1,4,0,8,4,-4,7,5,8,7,10,0,-4,-5,-8,4,11,7,9]

In [10]: t = pc( );print(tri_selection(new_l)) ;print(pc( )-t)
[-8,-5,-4,-4,0,0,1, 2, 4, 4, 4,5,7,7,7,8,8,9,10,11]
0.00032749999991210643
# on voit que l'ordre de grandeur est le meme pour le tri
    selection et le tri insertion.

In [11]: new2_l =
    [2,1,4,0,8,4,-4,7,5,8,7,10,0,-4,-5,-8,4,11,7,9]
In [12]: t = pc( ); print(new2_l.sort()) ; print(pc( ) - t)
None
0.0016777000000729458

In [13]: new2_l
Out[13]: [-8,-5,-4,-4,0,0,1,2,4,4,4,5,7,7,7,8,8,9,10,11]
In [14]: new2_l.sort()

In [15]: 0.0016777/0.00075
Out[15]: 2.2369333333333334
# un peu plus rapide avec la fonction sort
```

■ Complexité minimale d'un algorithme de tri

Les deux algorithmes de tri que nous avons étudiés ont une complexité $C_{max}(n)$ en $O(n^2)$. Peut-on faire mieux ? Oui : nous étudierons plus tard quand la récursivité sera traité des algorithmes dont la complexité maximale vérifie $C_{max}(n) \in O(n \ln n)$. Ces algorithmes seront optimaux au sens suivant :

Théorème

Un algorithme de tri procédant par comparaisons entre les clés des éléments à trier a, au mieux, une complexité $C_{max}(n) \in O(n \ln n)$.

La preuve utilise la notion d'arbre de décision.

Arbres binaires

Définition : *Un arbre binaire est un ensemble de* ***noeuds****, organisés de la façon suivante :*
• *un noeud et un seul n'est pointé par aucune flèche, c'est la* ***racine*** *de l'arbre ;*
• *de certains noeuds (appelés* ***noeuds internes****) partent une flèche gauche et une flèche droite, pointant chacune sur un noeud (ce sont les* ***fils*** *du noeud interne) ;*
• *des autres noeuds (appelés* ***feuilles****) ne part aucune flèche.*

Cette structure est utilisée pour représenter des données. Elle contient de l'information, stockée dans les feuilles et les noeuds internes (ce sont les **etiquettes** de l'arbre) et dans les flèches (ce sont les labels de l'arbre).

Exemple : Voici un exemple d'arbre binaire :

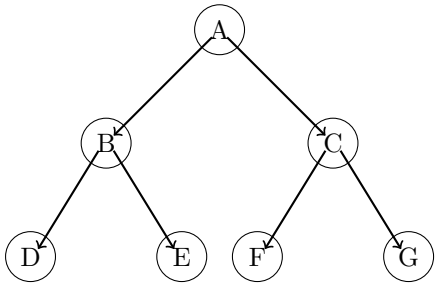


Figure 1

Cet arbre est formé de trois noeuds internes A, B et C, et de quatre feuilles D, E, F et G. Le noeud A est la racine de l'arbre ; le sous-arbre pointé par la flèche AB st le fils gauche du noeud A ; le sous-arbre pointé par la flèche AC est son fils droit. Ces deux sous-arbres ont pour père le noeud A.

Définition : *La* ***hauteur*** *d'un arbre est la longueur (ie nombre de flèches) maximale d'un chemin allant de la racine à une feuille.*

Quelle est la hauteur de l'arbre de l'exemple précédent ?

Exercice 03. *Représenter deux arbres de hauteur* 4*, le premier ayant* $4+1$ *feuilles et le second* 2^4 *feuilles.*

Proposition

Un arbre de hauteur n a donc au moins $n+1$ feuilles (cas où chaque fils gauche est une feuille par exemple) et au plus 2^n feuilles (cas où tous les noeuds sauf ceux du bas, ont 2 fils).

Preuve : représenter les cas extremes en s'inpirant de l'exercice précédent.

Arbre de décision

On désire faire un choix dans un ensemble E de cardinal $n \neq 0$. Un **arbre de décision** associé à ce choix est un arbre binaire tel que :
- la racine est étiquettée par l'ensemble E
- chaque feuille est étiquettée par un singleton inclus dans E

• chaque noeud qui n'est point une feuille est étiqueté par un sous-ensemble E_1 de E et ses deux fils par des sous-ensembles E_2 et E_3 de E_1 tels que $E_1 = E_2 \cup E_3$. Attention, E_2 et E_3 ne sont pas nécessairement disjoints.
• On adjoint à chaque noeud un test booléen et les deux flèches issues de ce noeud portent les labels T ou F pour $True$ ou $False$.

Exercice 04. *Faire un arbre de décision pour la recherche de $min(a, b, c)$, c'est-à-dire associé au choix du plus petit parmi les nombres a, b et c.*

Proposition

Un arbre de décision associé à un ensemble de cardinal n doit posséder n feuilles au moins ; sa hauteur h vérifie : $h \geqslant \log_2(n)$.

Utilisons maintenant un arbre de décision pour le problème du tri d'une liste l de taille n. En supposant les éléments deux à deux distincts, il y a $n!$ listes différentes contenant ces éléments donc $n!$ feuilles. Chacune de ces feuilles est constituée d'une permutation de la liste l. Trier la liste revient à trouver, parmi les $n!$ permutations de l, quelle est celle qui est triée. Notre arbre de décision a pour ensemble de référence l'ensemble E des permutations de l ; les tests associés à chaque noeud sont les tests comparant deux éléments du tableau. La hauteur minimale de l'arbre de décision est donc supérieure ou égale à

$$\log_2(n!) = \sum_{k=2}^{n} \log_2(k).$$

Or la croissance de ln permet d'écrire :

$$\forall k \geqslant 2, \int_{k-1}^{k} \ln t\, dt \leqslant \ln k \leqslant \int_{k}^{k+1} \ln t\, dt.$$

Et en sommant :

$$\sum_{k=2}^{n} \int_{k-1}^{k} \ln t\, dt \leqslant \sum_{k=2}^{n} \ln k \leqslant \sum_{k=2}^{n} \int_{k}^{k+1} \ln t\, dt.$$

C'est-à-dire :

$$\int_{1}^{n} \ln t\, dt \leqslant \ln(n!) \leqslant \int_{2}^{n+1} \ln t\, dt.$$

Il reste à intégrer :

$$n \ln n - n + 1 \leqslant \ln(n!) \leqslant (n+1)\ln(n+1) - 2(\ln 2 - 1).$$

Puis cela donne en divisant par $n \ln n$:

$$\frac{n \ln n - n + 1}{n \ln n} \leqslant \frac{\ln(n!)}{n \ln n} \leqslant \frac{(n+1)\ln(n+1) - 2(\ln 2 - 1)}{n \ln n}.$$

On voit rapidement que quand n tend vers $+\infty$, $\dfrac{n \ln n - n + 1}{n \ln n}$ et $\dfrac{(n+1)\ln(n+1) - 2(\ln 2 - 1)}{n \ln n}$ tendent vers 1. Donc d'après le théorème des Gendarmes,

$$\ln(n!) \sim n \ln n \Rightarrow \log_2(n!) \sim n \log_2(n).$$

Comme le nombre de comparaisons dans le pire des cas est égal à la hauteur de l'arbre de décision, nous avons démontré le théorème.

■ Tris sans comparaison

Les tris présentés auparavant sont des tris qui utilisent les comparaisons. Il existe d'autres types de tris par exemple par comptage, par paquets, par base. Présentons ici le tri par comptage.

On dispose d'une liste d'entiers naturels qui sont tous inférieurs à un entier naturel non num m qui est de l'ordre de n qui est la taille de la liste.

On commence par écrire une fonction **comptage**, d'arguments une liste **entiers** et un entier **m**, renvoyant une liste de longueur **m+1** telle que pour tout 'k de 0 à m, l'élément d'indice **k** a pour valeur le nombre d'occurrences de l'entier **k** dans la liste **entiers**. Pour cela, on crée une liste composée de 0 et de longueur $m+1$. Chaque élément de cette liste sert de compteur.

Commençons par une première version nommée **comptage**

```
In [1]: def comptage(entiers,m):
   ...:     compteurs = (m+1) * [0]
   ...:     for k in range(m+1):
   ...:         cpt = 0
   ...:         for p in entiers :
   ...:             if p == k :
   ...:                 cpt = cpt + 1
   ...:         compteurs[k] = cpt
   ...:     return compteurs
   ...:
```

Exercice 05 *Appliquer à la main* **comptage** *à la liste* **[1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2]** *avec* **m=3** *puis* **m=7** *puis* **m=17**. *Qu'obtient-on ?*

```
# SOLUTION :

In [15]: comptage([1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2],17)
Out[15]: [3, 4, 2, 0, 4, 1, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [16]: comptage([1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2],7)
Out[16]: [3, 4, 2, 0, 4, 1, 0, 2]

In [17]: comptage([1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2],3)
Out[17]: [3, 4, 2, 0]
```

Cet algorithme contient deux boucles imbriquées et son coût est de l'ordre $m \times n$ et donc quadratique en n car $m = O(n)$. Voici donc une version plus efficace, dont le coût en fonction de la longueur n de la liste est linéaire. On la note **new_comptage**.

```
In [5]:  def new_comptage(entiers,m):
   ...:     compteurs = (m+1) * [0]
   ...:     for k in entiers :
   ...:        compteurs[k] = compteurs[k] + 1
   ...:     return compteurs
```

Les valeurs des éléments de la liste **entiers** sont représentés par les indices de la liste **compteurs**. On en déduit une fonction **tri**, d'arguments une liste **entiers** et un entier **m**, renvoyant la liste tridans l'ordre croissant.

Exercice 06 *Appliquer à la main* **new_comptage** *à la liste* **[1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2]** *avec* **m=3** *puis* **m=7** *puis* **m=17** *et* **m=22**. *Qu'obtient-on ?*

```
# SOLUTION :
In [18]: new_comptage([1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2],17)
Out[18]: [3, 4, 2, 0, 4, 1, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [19]: new_comptage([1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2],7)
Out[19]: [3, 4, 2, 0, 4, 1, 0, 2]

In [20]: new_comptage([1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2],3)
Traceback (most recent call last):

  File "<ipython-input-20-ab76a1fab2fb>", line 1, in <module>
    new_comptage([1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2],3)

  File "<ipython-input-5-de8aff887c7b>", line 4, in new_comptage
    compteurs[k] = compteurs[k] + 1

IndexError: list index out of range
# que se passe t-il si l'on prend des negatifs ?
In [9] : new_comptage([-1,2,4,5,4,1,0,1,0,0,7,-1,4,4,7,2],22)
Out[9]: [3, 2, 2, 0, 4, 1, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 2]
```

Le nombre d'opérations de la fonction **new_comptage** est de l'ordre de n, la longueur de la liste à trier (car $m = O(n)$)) et dans la fonction **tri**, on construit une liste de même longueur. Le coût total est donc linéaire en la longueur de la liste.
Si l'entier m n'est pas donné, il suffit de déterminer le maximum de la liste. Le coût total ne change pas puisque la recherche d'un maximum a un coût linéaire en la longueur de la liste.

```
In [10]: def tri(entiers,m):
   ...:     cpts = new_comptage(entiers, m)
   ...:     triee = []
   ...:     for i in range(m+1):
   ...:         triee = triee + cpts[i] * [i]
   ...:     return triee
   ...:
```

```
In [21]: tri([1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2],17)
Out[21]: [0, 0, 0, 1, 1, 1, 1, 2, 2, 4, 4, 4, 4, 5, 7, 7]

In [22]: tri([1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2],22)
Out[22]: [0, 0, 0, 1, 1, 1, 1, 2, 2, 4, 4, 4, 4, 5, 7, 7]

In [23]: tri([1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2],7)
Out[23]: [0, 0, 0, 1, 1, 1, 1, 2, 2, 4, 4, 4, 4, 5, 7, 7]

In [24]: tri([1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2],3)
Traceback (most recent call last):

  File "<ipython-input-24-14d2f94d5f27>", line 1, in <module>
    tri([1,2,4,5,4,1,0,1,0,0,7,1,4,4,7,2],3)

  File "<ipython-input-9-42b405eef969>", line 2, in tri
    cpts = new_comptage(entiers, m)

  File "<ipython-input-5-de8aff887c7b>", line 4, in new_comptage
    compteurs[k] = compteurs[k] + 1

IndexError: list index out of range
```

Chapitre 8

Algorithm of Gauss

Objectifs

- **Les incontournables :**
  - savoir

Cours

Il s'agit de créer une fonction Python qui permet de résoudre un système linéaire avec la méthode de Gauss. Mais auparavant, on va faire fonctionner la syntaxe autour des matrices puis on va créer les opérations élémentaires classiques sur les lignes et colonnes de matrices.

Pour la culture, parlons de Gauß

Nom de naissance Johann Carl Friedrich Gauß
Naissance 30 avril 1777 Brunswick (Principauté de Brunswick-Wolfenbüttel)
Décès 23 février 1855 (à 77 ans) Göttingen (Royaume de Hanovre)
Nationalité : Allemand
Domaines : Astronomie, mathématiques, physique
Lieu de travail : Université de Göttingen
Diplôme : Université de Helmstedt

■ Les codes Python autour des matrices

Comment écrire puis manipuler des matrices avec Python
On travaille surtout avec `numpy` et son sous-module `numpy.linalg` entrevus dans un chapitre antérieur. Rappelons ce que l'on tape toujours au début.

```
In [1]: import numpy as np ; import numpy.linalg as alg
```

Dans la suite, soit $A \in \mathcal{M}_{n,p}(\mathbb{R})$, dont les lignes sont les listes $L_0, ..., L_{n-1}$, toutes de même taille p et les colonnes sont les listes $C_0, ..., C_{p-1}$ toutes de même taille n.
Attention, en Math, les indices partent conventionnellement de 1 ***mais en Python de*** 0 ***donc ici tous nos indices démarreront à*** 0 ***pour ne pas faire des mélanges.***

- Pour ***définir la matrice*** $A \in \mathcal{M}_{n,p}(\mathbb{R})$, sous `numpy`, on tape un code (en rentrant les listes de ligne) de la forme `-1A = np.array([L_0,...,L_n ])`.
- La commande `A.shape` donne la ***taille*** (n, p) de A. La commande `A.size` renvoie $n \times p$, `np.size(A,0)` (*resp.* `np.size(A,1)` renvoie n (*resp.* p).
- Le ***coefficient*** de A à l'intersection de la ligne L_i et colonne C_j est `A[i,j]`.
- Le code `A[i, :]` (ou aussi `A[i]`) est la ***ligne*** L_i (tableau à une dimension) de A.
- Le code `A[:, j]` est la ***colonne*** C_j (tableau à une dimension) de A.
- Le code `A[i:j, k:l]` renvoie la ***sous-matrice*** de la ligne L_i à la ligne L_{j-1} et de la colonne C_k à la colonne C_{l-1}.
- Le code `p.zeros((n,p))` renvoie la ***matrice nulle*** de $\mathcal{M}_{n,p}(\mathbb{R})$ et `np.eye(n)` renvoie I_n, la matrice identité d'ordre n.
- Le code `np.ones((n,p))` renvoie la matrice Attila (composée que de 1) de $\mathcal{M}_{n,p}(\mathbb{R})$
- Le code `np.diag(L)` renvoie la ***matrice carrée diagonale*** dont les coefficients de la diagonale principale sont les éléments de la liste L dans l'ordre.

- Si l'on veut réunir des matrices A et B (donc les ***concatener***), en tapant le code `np.concatenate((A,B),axis=0)`, on renvoie $\begin{pmatrix} A \\ B \end{pmatrix}$ et si l'on fait le code `np.concatenate((A,B),axis=1)`, on renvoie $\begin{pmatrix} A & | & B \end{pmatrix}$.
  Ce sont des codes utiles pour appliquer la méthode de Gauss-Jordan.
- Pour ***ajouter*** les matrices A et B, on tape `A+B` tout simplement et pour renvoyer `3A` par exemple, on tape `3*A`
- Pour ***effectuer le produit matriciel*** $A \times B$, on tape `np.dot(A,B)` ou encore `A.dot(B)`. Pour calculer le produit de trois matrices A, B et C, on tape `A.dot(B).dot(C)` Pour calculer A^n, on peut utiliser la fonction `matrix_power` en tapant `alg.matrix_power(A,n)`
- Pour calculer la ***transposée*** de A, on tape `np.transpose(A)`
  (on peut aussi utiliser l'attribut `T` en écrivant `A.T`
  Remarque : une application de `np.transpose` est de convertir un tableau-ligne en tableau-colonne (ou le contraire). Cela peut être utile.
- Pour calculer directement le ***déterminant*** de A (si $n = p$), on tape `alg.det(A)`
- Pour calculer directement le ***rang*** de A, on tape `alg.matrix_rank(A)`
- Pour calculer directement la ***trace*** de A, on tape `np.trace(A)`
- Pour calculer ***l'inverse*** de A (si $n = p$), on tape `alg.inv(A)`
- Pour appliquer une fonction f à tous les coefficients de $A = (a_{i,j})$, on tape `f(A)`

Exercice 01 *: On considère* $A = \begin{pmatrix} 1 & -1 & 0 \\ -2 & 4 & 1 \\ 0 & 3 & -5 \end{pmatrix}$ *et* $B = \begin{pmatrix} 1 & 0 \\ 1 & 3 \\ -1 & 2 \end{pmatrix}$.

1. *Rentrer les matrices A et B sous Python.*
2. *Avec Python, donner le nombre de lignes de A et de colonnes de B.*
3. *Renvoyer la ligne L_3 et la colonne C_2 de A.*
Extraire la sous-matrice issue de A en supprimant L_3 et C_3.
4. *Concatener ensemble B et une colonne de trois zéros. On obtient alors une matrice carrée d'ordre 3 que l'on appellera C.*
5. *Comment renvoyer $A + C$ et AC ?*
6. *Calculer avec Python A^{-1} et C^{-1}. que remarque t-on ?*
7. *Comment calculer A^7 avec Python ?*
8. *Construire la fonction $f : x \mapsto e^{2x}$ en Python puis écrire le code qui permet de renvoyer la matrice dont tous les coefficients sont les images par f des coefficients dev A.*

■ Les opérations élémentaires sur les lignes et colonnes en code Python

Si i et j sont deux entiers distincts dans $[\![0, n-1]\!]$, on a les opérations sur les lignes :

$$\boxed{L_i \leftarrow L_i + \lambda L_j, \quad L_i \leftarrow \lambda L_i, \quad L_i \leftrightarrow L_j.}$$

Si i et j sont deux entiers distincts dans $[\![0, p-1]\!]$, on a les opérations sur les colonnes :

$$\boxed{C_i \leftarrow C_i + \lambda C_j, \quad C_i \leftarrow \lambda C_i, \quad C_i \leftrightarrow C_j.}$$

Comment écrire en langage Python les opérations élémentaires classiques sur les lignes de matrices
Il s'agit de créer les fonctions $L_i \leftarrow L_i + xL_j$, $L_i \leftarrow xL_i$ et $L_i \leftrightarrow L_j$.

- Pour ***effectuer la transvection*** $L_i \leftarrow L_i + xL_j$ sur une matrice A, on tapera :

```
# VERSION LONGUE
In [23]: def old_transvecligne(A,i,j,x) :
    ...:     p = np.size(A,1)
    ...:     for m in range(p) :
    ...:         A[i,m] = A[i,m] + x*A[j,m]
    ...:     return A
    ...:
# VERSION COURTE
In [26]: def transvecligne(A,i,j,x):
    ...:     A[i,:] = A[i,:] + x*A[j,:]
    ...:     return A
```

Exercice 02 *: Appliquer* $L_1 \leftarrow L_1 - L_2$ *à* $A = \begin{pmatrix} 1 & -1 & 0 \\ -2 & 4 & 1 \\ 0 & 3 & -5 \end{pmatrix}$ *d'abord avec* **`old_transvecligne`** *puis* **`transvecligne`**. *Attention, la matrice* A *après l'action de* **`old_transvecligne`** *sera transformée. Vérifier à la main.*

- Pour ***effectuer la dilatation*** $L_i \leftarrow xL_i$ sur A, on tape (version courte directe) :

```
In [29]: def dilatligne(A,i,x):
    ...:     A[i,:] = x * A[i,:]
    ...:     return A
# Par exemple, dilatons L1 de la matrice Attila
In [30]: dilatligne(np.ones((3,5)),1,2024)
Out[30]:
array([[1.000e+00, 1.000e+00, 1.000e+00, 1.000e+00, 1.000e+00],
       [2.024e+03, 2.024e+03, 2.024e+03, 2.024e+03, 2.024e+03],
       [1.000e+00, 1.000e+00, 1.000e+00, 1.000e+00, 1.000e+00]])
```

- Pour ***effectuer la permutation*** $L_i \leftrightarrow L_j$ sur A, on tape :

```
# VERSION LONGUE
In [34]: def old_permutligne(A,i,j):
    ...:     p = np.size(A,1)
    ...:     for m in range(p):
    ...:         temp=A[i,m];A[i,m]=A[j,m];A[j,m]=temp
    ...:     return A
# VERSION COURTE
In [35]: def permutligne(A,i,j):
    ...:     A[i,:] , A[j,:] = np.copy(A[j,:]) , np.copy(A[i,:])
    ...:     return A
```

Vous avez remarqué que dans la version alternative de **old_permutligne**, on a utilisé la fonction **np.copy** du module **numpy** (d'alias *np*). En effet, si l'on n'utilise pas **np.copy**, on se retrouverait avec une matrice ayant deux lignes égales.
Faisons fonctionner, juste pour le fun.

```
In [36]:  A = np.array([[1,-1,0],[-2,4,1],[0,3,-5]])
In [37]: old_permutligne(A,0,2)
Out[37]:
array([[ 0,  3, -5],
       [-2,  4,  1],
       [ 1, -1,  0]])
In [38]: permutligne(A,1,2)
Out[38]:
array([[ 0,  3, -5],
       [ 1, -1,  0],
       [-2,  4,  1]])
```

Exercice 03. *Écrire les fonctions Python qui correspondent aux trois opérations sur les colonnes que l'on notera* **transveccolonne**, **dilatcolonne** *et* **permutcolonne**. *On adaptera les versions courtes des fonctions sur les lignes écrites plus haut. Appliquer alors à transformer la matrice :* $A = \begin{pmatrix} 1 & -1 & 3 \\ 0 & 1 & 2 \\ -1 & -2 & 0 \end{pmatrix}$ *en* I_3 *en utilisant un certain nombre des six fonctions Python, de la façon dont cela vous semble nécessaire.*
Pour ***effectuer la transvection*** $C_i \leftarrow C_i + kC_j$ sur A, on tape :

```
In [39]: def transveccolonne(A,i,j,k) :
             A[:,i] += k*A[:,j]
             return A
```

Pour ***effectuer la dilatation*** $C_i \leftarrow kC_i$ sur A, on tape :

```
In [40]: def dilatcolonne(A,i,x) :
             A[:,i] = x * A[:,i ]
             return A
```

Pour ***effectuer la permutation*** $C_i \leftrightarrow C_j$ sur A, on tape :

```
In [41]: def permutcolonne(A,i,j) :
             A[:,i],A[:,j] = np.copy( A[:,j]),np.copy(A[:,i])
             return A
```

Application à l'algorithme de Gauss-Jordan pour inverser une matrice

Cette méthode permet aussi à la fois de prouver l'inversibilité et de trouver l'inverse. Matriciellement, les opérations élémentaires sur les lignes correspondent à la multiplication à gauche par un certain nombre de matrices $Q_i \in GL_p(\mathbb{K})$ de la matrice A pour obtenir $Q_k Q_{k-1} ... Q_1 A = I_p$, ce qui montre déjà que A est inversible (car elle possède un inverse à gauche) puis en multipliant à droite chacun des membres de cette égalité par A^{-1}, on obtient $A^{-1} = Q_k Q_{k-1} ... Q_1$. Cela justifie le fait que A^{-1} se retrouve après opérations élémentaires sur les lignes dans la partie droite de la matrice à p lignes et $2p$ colonnes obtenue.

- on écrit d'abord la matrice à p lignes et $2p$ colonnes schématisée par $\left(\begin{array}{c|c} A & I_p \end{array} \right)$, le symbole | étant juste une barrière virtuelle qui permet de distinguer les deux parties importantes de la matrice;
- on transforme cette matrice par opérations élémentaires **sur les lignes** (sans se soucier de la barrière virtuelle) pour aboutir à la matrice schématisée par $\left(\begin{array}{c|c} I_p & A^{-1} \end{array} \right)$.

Exercice 04 *: inverser avec Python* $A = \begin{pmatrix} 2 & 2 & 3 \\ 1 & 1 & 4 \\ 1 & -2 & 1 \end{pmatrix}$. *On commencera par concaténer* A *et* I_3 *et on utilisera les fonctions Pythons qui font les opérations élémentaires sur les lignes. À la fin, on déconcaténera.*

■ Algorithme du pivot de Gauss en code Python

Comment résoudre un système linéaire $AX = B$ **?**

- On peut appliquer une ***méthode directe*** lorsque A est carrée et inversible en employant la fonction **`solve`** du sous-module **`numpy.linalg`**.
  On tape : **`alg.solve(A,B)`**

  Nous obtenons X sous forme d'un tableau-ligne.
  Voir le premier exemple qui suit pour la mise en forme.
- On peut appliquer la ***méthode du pivot de Gauss*** et les fonctions qui ont été construites dans le paragraphe précédent. Voir la suite pour le programme Python correspondant.

Exemple 01. *Utilisons le module* **`numpy`** *et sa fonction* **`solve`** *pour résoudre :*

$$\left\{ \begin{array}{rcl} 10^{-20}x + y & = & 1 \\ x + y & = & 2 \end{array} \right. .$$

```
In [41]: A = np.array([[10**(-20),1],[1,1]])
In [42]: B = np.array([[1],[2]])
In [43]: alg.solve(A,B)
Out[42]:
         array([[1.],[1.]])
```

Il est clair que le résultat est "arrondi".

Passons à l'algorithme de Gauss. Nous allons résoudre le système linéaire de matrice A (carrée et inversible) et de second membre B par la méthode du pivot de Gauss. Bien entendu, si A s'avère non inversible, le programme renvoie un message d'erreur.
On peut sinon par exemple faire un test en tête de procédure en demandant le calcul du déterminant. S'il est nul, on renvoie un message qui demande de transformer notre système, par exemple en faisant passer des inconnues dans la matrice colonne B et A est ramené à une de ses sous-matrices inversibles. Nous laissons de côté un tel test pour ne pas surcharger le programme.

L'algorithme de Gauss (dit aussi du pivot de Gauss) est basé sur des opérations élémentaires sur les lignes (qui ne modifient pas le rang de A ni les inconnues).

On supposera ici que pour tout i, `A[i,i]` $\neq 0$.
La première chose à faire est de concatèner A et B en $D = \left(\begin{array}{c|c} A & B \end{array}\right)$.

Le coefficient `D[0,0]` est appelé premier pivot.
Pour i variant de 1 à $n-1$, où `n = np.size(A,0)`, on retranche à L_i la ligne L_0 multipliée par `D[i,0]/D[0,0]`
On obtient donc une nouvelle colonne C_0 avec `D[0,0]` et que des 0.
Puis on considère le nouveau `D[1,1]` que l'on suppose encore non nul et on refait le même type d'opérations. On met tout ça dans une boucle `for j in range(n-1)`
À la fin, la matrice AD devient triangulaire supérieure. À ce niveau, on peut en déduire en remontant la valeur des inconnues. La dernière ligne donne la valeur de la dernière inconnue, on remplace dans l'avant dernière équation. Ainsi de suite...
On peut aussi considèrer `D[n-1,n-1]` comme nouveau pivot et refaire l'algorithme du pivot de Gauss inversé (avec une boucle `for j in range(n-1,-1,-1)`)
À la fin, D devient une matrice diagonale (sauf sa dernière colonne) et en divisant par les coefficients diagonaux, ligne par ligne, la nouvelle dernière colonne de D sont les valeurs des inconnues.

Passons au programme Python. On crée une fonction appelée `systLin` qui résout le système linéaire de matrice A (carrée et inversible) et de second membre B par la méthode du pivot de Gauss complète c'est-à-dire que $\left(\begin{array}{c|c} A & B \end{array}\right)$ devient $\left(\begin{array}{c|c} A' & B' \end{array}\right)$, où A' est diagonale et est la dernière modification de A et B' est la dernière modification de B qui contient les solutions à un coefficient multiplicatif près. Il reste à utiliser une des opérations élémentaires pour avoir les solutions.
Les tableaux A et B seront les arguments de `systLin`.
La fonction ne devra modifier ni la matrice A, ni la matrice B (il faudra en faire des copies, sur lesquelles on fera les opérations en posant `A1 = np.copy(A)` et `B1 = np.copy(B)`).
On pourra utiliser les fonctions Python correspondantes aux opérations élémentaires c'est-à-dire plus précisément `transvecligne` dans les deux boucles doubles. En sortie des dernières boucles doubles, on récupère `D[:,n]` qui est la dernière colonne de `D` sous forme d'une liste. Puis le code `np.array([D[:,n]])` transforme `D[:,n]` en matrice ligne. On appelle `NewB1` cette matrice ligne et on utilise `dilatcolonne` pour diviser toutes les colonnes C_s (donc ses coefficients) de `NewB1` par `D[s,s]`. On notera `sol` la matrice ligne solution.
Exemple 02: *on testera* ***`systLin`*** *qui suit sur le système* :

$$\left\{\begin{array}{rcl} x-y & = & 1 \\ -2x+4y+z & = & 4 \\ 3y-5z & = & 0 \end{array}\right. .$$

```
In [1]: def systLin(A,B) :
            # on fait des copies de A et de B
            A1 = np.copy(A); B1 = np.copy(B)
            # on concatene ensemble A1 et B1 dans D
            D = np.concatenate((A1,B1),axis=1)
            print(D)
            n = np.size(A1,0)
            # On transforme D en une matrice triang sup
            for j in range(n-1) :
                for i in range(j+1,n) :
                    print('le pivot est :',D[j,j])
                    c =  D[i,j]/D[j,j]
                    print('on enleve a la',i,'eme ligne',c,'fois la',j,'
                        eme ligne')
                    D = transvecligne(D,i,j,-c)
                    print(D)
            # on transforme D (sauf derniere col) en matrice diag
            for j in range(n-1,0,-1) :
                for i in range(j-1,-1,-1) :
                    c = D[i,j]/D[j,j]
                    print('on enleve a la',i,'eme ligne',c,'fois la',j,'
                        eme ligne')
                    D = transvecligne(D,i,j,-c)
                    print(D)
            # On transforme la derniere colonne de D en matrice ligne
            NewB1 = np.array([D[:,n]])
            # On divise les lignes de NewB1 par les pivots successifs
            for s in range(n):
                    Sol = dilatcolonne(NewB1,s,1./D[s,s])
            # On retourne alors les solutions du systeme
            return Sol
```

Warning : Attention, on rentre les coefficients des lignes de `A` et `B` sous forme de flottants (donc en rajoutant un .). En effet, sinon, il y aura des arrondis dans les divisions qui conduiront à un résultat erroné.

```
# PASSONS A EXEMPLE 02
In [2]: A = np.array([[1.,-1.,0.],[-2.,4.,1.],[0.,3.,-5.]])
In [3]: B = np.array([[1.],[4.],[0.]])
In [4]: systLin(A,B)
```

Et découvrons ensuite le résultat.

```
# on donne la matrice D produit du concatenage
[[ 1. -1.  0.  1.]
 [-2.  4.  1.  4.]
 [ 0.  3. -5.  0.]]

le pivot est : 1.0
on enleve a la 1 eme ligne -2.0 fois la  0 eme ligne
# on a fait L1 <- L1 + 2 L0
[[ 1. -1.  0.  1.]
 [ 0.  2.  1.  6.]
 [ 0.  3. -5.  0.]]
le pivot est : 1.0
on enleve a la 2 eme ligne 0.0 fois la  0 eme ligne
# on a fait L2 <- L2 donc rien du tout
[[ 1. -1.  0.  1.]
 [ 0.  2.  1.  6.]
 [ 0.  3. -5.  0.]]

le pivot est : 2.0
on enleve a la 2 eme ligne 1.5 fois la  1 eme ligne
# on a fait L2 <- L2 -3/2 L1
[[ 1.  -1.   0.   1. ]
 [ 0.   2.   1.   6. ]
 [ 0.   0.  -6.5 -9. ]]

# on attaque la deuxieme double boucle celle de la remontada

on enleve a la 1 eme ligne -0.15384615384615385 fois la  2 eme ligne
[[ 1.         -1.          0.          1.        ]
 [ 0.          2.          0.          4.61538462]
 [ 0.          0.         -6.5        -9.        ]]

on enleve a la 0 eme ligne -0.0 fois la  2 eme ligne
[[ 1.         -1.          0.          1.        ]
 [ 0.          2.          0.          4.61538462]
 [ 0.          0.         -6.5        -9.        ]]

on enleve a la 0 eme ligne -0.5 fois la  1 eme ligne
[[ 1.          0.          0.          3.30769231]
 [ 0.          2.          0.          4.61538462]
 [ 0.          0.         -6.5        -9.        ]]

# la matrice newB1 est :
[[ 3.30769231  4.61538462 -9.        ]]
# puis on retourne sol
array([[3.30769231, 2.30769231, 1.38461538]])
```

Intéressons-nous maintenant à la **complexité algorithmique de la méthode du pivot de Gauss** en posant n la taille de la matrice A.
A l'intérieur des deux premières boucles imbriquées, on a une division et 2 appels à la fonction `transvecligne`, ce qui fait 2n+1 opérations au plus. Comme les deux boucles imbriquées sont un $O(n^2)$, on a donc une complexité en $O(n^3)$.
A l'intérieur des deux autres boucles imbriquées, on a encore une division et 2 appels à `transvecligne` ce qui fait 2n+1 opérations au plus. Comme les deux boucles imbriquées sont un $O(n^2)$, on a donc encore une complexité en $O(n^3)$. Et on a un appel à `dilatligne` dans la boucle extérieure, ce qui fait un $O(n^2)$.
Donc en sommant $O(n^3) + O(n^3) + O(n^2)$, on a une complixité totale de $O(n^3)$.

Pour votre culture, l'algorithme de Strassen, qui est en $O(n^{2.807})$ a une meilleure complexité algorithmique asymptotique.

Programme du pivot de Gauss amélioré

Commençons par illustrer avec un nouveau système.

Exercice 05. *Tester* `systLin` *sur le système suivant :* $\begin{cases} x+3y+2z-5t &= 1 \\ x-4y-2z+6t &= -2 \\ 3x+y-2z+t &= 3 \\ -2x+y-4z+3t &= 2 \end{cases}$.

Que remarque t-on ?

On remarque que le dernier pivot de la première double boucle est un 0. Et donc on divise par 0. Il apparaît ensuite le code `nan` qui signifie : not a number ce qui pour Python est une forme polie de dire que c'est une valeur illégale. Puis apparaît aussi des `inf` qui signifie plus clairement des nombres infinis.

Revenons au cas général. Pour minimiser les erreurs d'arrondi et s'affranchir d'une situation de pivot nul, il faut déterminer le plus grand pivot possible en valeur absolue sur la partie de colonne considérée puis permuter les lignes. C'est la recherche partielle du pivot. Pour commencer donnons une procédure qui donne l'indice du plus grand élément d'une liste en valeur absolue.

```
In [1]: def indice_max_abs(L):
    ...:     """prend en argument une liste ou un tableau et renvoie
       indice de plus grand element de L en valeur absolue"""
    ...:     maxi = abs(L[0])
    ...:     index = 0
    ...:     for i in range(len(L)):
    ...:         if abs(L[i]) > maxi :
    ...:             maxi = abs(L[i])
    ...:             index = i
    ...:     return index
In [2]: indice_max_abs([-1,2,3,-4,1])
Out[2]:
       3
```

Nous allons donc créer la fonction Python nommée `NEW_systLin` qui correspondra à la fonction `systLin` mais en y ajoutant la recherche du pivot le plus grand en valeur absolue sur chaque colonne dans la descente vers un système triangulaire.

Nous allons donc utiliser **indice_max_abs** mais uniquement sur la première double boucle, après **for j in range(n-1) :**, on y ajoute la commande **index = indice_max_abs(D[j:n,j])+j** qui cherchera donc le plus grand coefficient de **D** en valeur absolue de la ligne L_j à la ligne L_{n-1} sur la colonne C_j. Comme **D[j:n,j])** est une liste, son indexation interne commence à 0 et donc quand on a trouvé l'indice du plus grand élément en valeur absolue de **D[j:n,j])**, on doit lui rajouter **j** pour revenir à l'indexation des lignes de **D**.
Puis on récupère cet indice en le nommant **index**. Puis on permute L_j et L_{index} de **D**

```
In [41]: def NEW_systLin(A,B) :
             # on fait des copies de A et de B
             A1 = np.copy(A); B1 = np.copy(B)
             # on concatene ensemble A1 et B1 dans D
             D = np.concatenate((A1,B1),axis=1)
             n = np.size(A1,0);print(D)
             # On transforme D en une matrice triang sup
             print("On transforme en systeme triang sup :")
             for j in range(n-1) :
                 print("On passe a la colonne C",j)
                 index = indice_max_abs(D[j:n,j]) + j
                 print("indice ligne du coeff le plus grand:",index)
                 D = permutligne(D,j,index); print(D)
                 for i in range(j+1,n) :
                     print('le pivot est :',D[j,j])
                     c = D[i,j]/D[j,j]
                     print('on enleve a la',i,'eme ligne', c,'fois la',j
                         ,'eme ligne')
                     D = transvecligne(D,i,j,-c)
                     print(D)
             # on transforme D (sauf derniere col) en matrice diag
             print("On transforme en systeme diagonal :")
             for j in range(n-1,0,-1) :
                 for i in range(j-1,-1,-1) :
                     c = D[i,j]/D[j,j]
                     print('on enleve a la',i, 'eme ligne',c,'fois la',j
                         ,'eme ligne')
                     D = transvecligne(D,i,j,-c)
                     print(D)
             # On transforme la derniere colonne de D en matrice ligne
             NewB1 = np.array([D[:,n]]) ; print(NewB1)
             # On divise les lignes de NewB1 par les pivots successifs
             for s in range(n):
                     Sol = dilatcolonne(NewB1,s,1./D[s,s])
             # On retourne alors les solutions du systeme
             return Sol
```

Exemple 02 bis. *Testons* **NEW_systLin** *sur le système de l'exemple 02.*

```
In [40]: A = np.array([[1.,-1.,0.],[-2.,4.,1.],[0.,3.,-5.]])
In [41]: B = np.array([[1.],[4.],[0.]])
In [42]: NEW_systLin(A,B)
[[ 1. -1.  0.  1.]
 [-2.  4.  1.  4.]
 [ 0.  3. -5.  0.]]
On transforme en systeme triang sup :
On passe a la colonne C 0
indice ligne du coeff le plus grand: 1
[[-2.  4.  1.  4.]
 [ 1. -1.  0.  1.]
 [ 0.  3. -5.  0.]]
le pivot est : -2.0
on enleve a la 1 eme ligne -0.5 fois la 0 eme ligne
[[-2.   4.   1.   4. ]
 [ 0.   1.   0.5  3. ]
 [ 0.   3.  -5.   0. ]]
le pivot est : -2.0
on enleve a la 2 eme ligne -0.0 fois la 0 eme ligne
[[-2.   4.   1.   4. ]
 [ 0.   1.   0.5  3. ]
 [ 0.   3.  -5.   0. ]]
On passe a la colonne C 1
indice ligne du coeff le plus grand: 2
[[-2.   4.   1.   4. ]
 [ 0.   3.  -5.   0. ]
 [ 0.   1.   0.5  3. ]]
le pivot est : 3.0
on enleve a la 2 eme ligne 0.3333333333333333 fois la 1 eme ligne
[[-2.          4.          1.          4.        ]
 [ 0.          3.         -5.          0.        ]
 [ 0.          0.          2.16666667  3.        ]]
On transforme en systeme diagonal :
on enleve a la 1 eme ligne -2.307692307692308 fois la 2 eme ligne
[[-2.          4.          1.          4.        ]
 [ 0.          3.          0.          6.92307692]
 [ 0.          0.          2.16666667  3.        ]]
on enleve a la 0 eme ligne 0.46153846153846156 fois la 2 eme ligne
[[-2.          4.          0.          2.61538462]
 [ 0.          3.          0.          6.92307692]
 [ 0.          0.          2.16666667  3.        ]]
on enleve a la 0 eme ligne 1.3333333333333333 fois la 1 eme ligne
[[-2.          0.          0.         -6.61538462]
 [ 0.          3.          0.          6.92307692]
 [ 0.          0.          2.16666667  3.        ]]
[[-6.61538462  6.92307692  3.        ]]
Out[42]: array([[3.30769231, 2.30769231, 1.38461538]])
```

Exercice 06. *Tester* **`NEW_systLin`** *sur le système de l'exercice 05.*

Correction des exercices

```
# CORRECTION EXERCICE 01
In [2]:  A = np.array([[1,-1,0],[-2,4,1],[0,3,-5]])
In [3]: A
Out[3]: array([[ 1, -1,  0],
       [-2,  4,  1],
       [ 0,  3, -5]])
In [4]: B = np.array([[1,0],[1,3],[-1,2]])
In [15]: A[2,:] ;  A[:,1] ;
Out[15]: array([ 0,  3, -5]) array([-1,  4,  3])
In [21]: A[0:1,0:1]
Out[21]: array([[1]])
In [22]: A[0:2,0:2]
Out[22]: array([[ 1, -1], [-2,  4]])
In [7]: C = np.concatenate((B,np.zeros((3,1))), axis = 1)
In [8]: C
Out[8]: array([[ 1.,  0.,  0.],
       [ 1.,  3.,  0.],
       [-1.,  2.,  0.]])
In [10]: A+C
Out[10]: array([[ 2., -1.,  0.],
       [-1.,  7.,  1.],
       [-1.,  5., -5.]])
In [11]: A.dot(C)
Out[11]: array([[ 0., -3.,  0.], [ 1., 14.,  0.],[ 8., -1.,  0.]])
In [12]: alg.inv(A)
Out[12]: array([[ 1.76923077,  0.38461538,  0.07692308],
       [ 0.76923077,  0.38461538,  0.07692308],
       [ 0.46153846,  0.23076923, -0.15384615]])
In [13]: alg.inv(C)
# CA GUEULE !
In [14]: alg.matrix_power(A,7)
Out[14]: array([[   6929,  -14365,     676],
         [ -28730,   47996,   18421],
         [   4056,   55263, -116441]])
In [18]: def f(x):
    ...:     return np.exp(2*x)
In [20]: f(A)
Out[20]: array([[7.38905610e+00, 1.35335283e-01, 1.00000000e+00],
       [1.83156389e-02, 2.98095799e+03, 7.38905610e+00],
       [1.00000000e+00, 4.03428793e+02, 4.53999298e-05]])
```

```
# SOLUTION EXERCICE 02
In [24]: A
Out[24]:
array([[ 1, -1,  0],
       [-2,  4,  1],
       [ 0,  3, -5]])

In [25]: transvecligne(A,1,2,-1)
Out[25]:
array([[ 1, -1,  0],
       [-2,  1,  6],
       [ 0,  3, -5]])
In [27]: A
Out[27]:
array([[ 1, -1,  0],
       [-2,  1,  6],
       [ 0,  3, -5]])

In [28]: new_transvecligne(A,1,2,-1)
Out[28]:
array([[ 1, -1,  0],
       [-2, -2, 11],
       [ 0,  3, -5]])
```

```
# SOLUTION EXERCICE 03
In [42]: A = np.array([[1.,-1.,3.],[0.,1.,2.],[-1.,-2.,0.]])

In [43]: transveccolonne(A,1,0,1) , transveccolonne(A,2,0,-3)

In [44]: transveccolonne(A,2,1,-2), dilatcolonne(A,2,1/9)

In [45]: transveccolonne(A,1,2,3) , transveccolonne(A,0,2,1)
Out[45]:
       array([[1,0,0],[0,1,0],[0,0,1]])
```

On a effectué les opérations élémentaires suivantes dans l'ordre :

$C_1 \leftarrow C_1+C_0$, $C_2 \leftarrow C_2-3C_0$, $C_2 \leftarrow C_2-2C_1$, $C_2 \leftarrow \frac{1}{9}C_2$, $C_1 \leftarrow C_1+3C_2$, $C_0 \leftarrow C_0+C_2$.

```
# CORRECTION EXERCICE 04
# On rentre A et on concatene A et I3 en C
In [6]: A = np.array([[2,2,3],[1,1,4],[1,-2,1]])
In [8]: C = np.concatenate((A,np.eye(3)),axis = 1)
In [9]: C
Out[9]: array([[ 2.,  2.,  3.,  1.,  0.,  0.],
       [ 1.,  1.,  4.,  0.,  1.,  0.],
       [ 1., -2.,  1.,  0.,  0.,  1.]])
```

```
# C0 <-> C1
In [10]: permutligne(C,0,1)
Out[10]: array([[ 1.,  1.,  4.,  0.,  1.,  0.],
       [ 2.,  2.,  3.,  1.,  0.,  0.],
       [ 1., -2.,  1.,  0.,  0.,  1.]])
# L1 <- L1 - 2 L0 et L2 <- L2 - L0
In [11]: transvecligne(C,1,0,-2);transvecligne(C,2,0,-1)
Out[11]: array([[ 1.,  1.,  4.,  0.,  1.,  0.],
       [ 0.,  0., -5.,  1., -2.,  0.],
       [ 0., -3., -3.,  0., -1.,  1.]])
# C1 <-> C2
In [12]: permutligne(C,1,2)
Out[12]: array([[ 1.,  1.,  4.,  0.,  1.,  0.],
       [ 0., -3., -3.,  0., -1.,  1.],
       [ 0.,  0., -5.,  1., -2.,  0.]])
# L1 <- -1/3 L1 et L2 <- -1/5 L2
In [13]: dilatligne(C,1,-1/3);dilatligne(C,2,-1/5)
Out[13]: array([[ 1. ,  1. ,  4. ,  0. ,  1. ,  0. ],
       [-0. ,  1. ,  1. , -0. ,  0.33333333, -0.33333333],
       [-0. , -0. ,  1. , -0.2 ,  0.4 , -0. ]])
# L1 <- L1 - L2 et L0 <- L0 - 4 L2
In [14]: transvecligne(C,1,2,-1);transvecligne(C,0,2,-4)
Out[14]: array([[ 1. ,  1. ,  0. ,  0.8 , -0.6 ,  0. ],
       [ 0. ,  1. ,  0. ,  0.2 , -0.06666667, -0.33333333],
       [-0. , -0. ,  1. , -0.2 ,  0.4 , -0. ]])
# L0 <- L0 - L1
In [15]: transvecligne(C,0,1,-1)
Out[15]: array([[ 1. ,  0. ,  0. ,  0.6, -0.53333333,  0.33333333],
       [ 0. ,  1. ,  0. ,  0.2 , -0.06666667, -0.33333333],
       [-0. , -0. ,  1. , -0.2, 0.4 , -0. ]])
```

```
# Il reste a deconcatener
In [17]: C[0:3,3:6]
Out[17]:
array([[ 0.6        , -0.53333333,  0.33333333],
       [ 0.2        , -0.06666667, -0.33333333],
       [-0.2        ,  0.4       , -0.        ]])
# on remarque :
In [18]: -8/15 ; -1/15
Out[18]: -0.5333333333333333    -0.06666666666666667
```

On aboutit à la matrice C finale :

$$\left(\begin{array}{ccc|cc} 1 & 0 & 0 & 9/15 & -8/15 & 5/15 \\ 0 & 1 & 0 & 3/15 & -1/15 & -5/15 \\ 0 & 0 & 1 & -3/15 & 6/15 & 0 \end{array}\right).$$

```
# EXERCICE 05 CORRECTION
In [23]: B = np.array([[1.],[-2.],[3.],[2.]])
In [24]: A = np.array
    ([[1.,3.,2.,-5.],[1.,-4.,-2.,6.],[3.,1.,-2.,1.],[-2.,1.,-4.,3.]])
In [25]: systLin(A,B)
[[ 1.  3.  2. -5.  1.]
 [ 1. -4. -2.  6. -2.]
 [ 3.  1. -2.  1.  3.]
 [-2.  1. -4.  3.  2.]]
le pivot est : 1.0
on enleve    la 1  me  ligne 1.0 fois la  0  me  ligne
[[ 1.  3.  2. -5.  1.]
 [ 0. -7. -4. 11. -3.]
 [ 3.  1. -2.  1.  3.]
 [-2.  1. -4.  3.  2.]]
le pivot est : 1.0
on enleve    la 2  me  ligne 3.0 fois la  0  me  ligne
[[ 1.  3.  2. -5.  1.]
 [ 0. -7. -4. 11. -3.]
 [ 0. -8. -8. 16.  0.]
 [-2.  1. -4.  3.  2.]]
le pivot est : 1.0
on enleve    la 3  me  ligne -2.0 fois la  0  me  ligne
[[ 1.  3.  2. -5.  1.]
 [ 0. -7. -4. 11. -3.]
 [ 0. -8. -8. 16.  0.]
 [ 0.  7.  0. -7.  4.]]
le pivot est : -7.0
```

```
# ON CONTINUE
on enleve    la 2  me  ligne 1.1428571428571428 fois la  1  me  ligne
[[ 1.          3.          2.         -5.          1.        ]
 [ 0.         -7.         -4.         11.         -3.        ]
 [ 0.          0.         -3.42857143  3.42857143  3.42857143]
 [ 0.          7.          0.         -7.          4.        ]]
le pivot est : -7.0
on enleve    la 3  me  ligne -1.0 fois la  1  me  ligne
[[ 1.          3.          2.         -5.          1.        ]
 [ 0.         -7.         -4.         11.         -3.        ]
 [ 0.          0.         -3.42857143  3.42857143  3.42857143]
 [ 0.          0.         -4.          4.          1.        ]]
le pivot est : -3.428571428571429
on enleve    la 3  me  ligne 1.1666666666666665 fois la  2  me  ligne
[[ 1.          3.          2.         -5.          1.        ]
 [ 0.         -7.         -4.         11.         -3.        ]
 [ 0.          0.         -3.42857143  3.42857143  3.42857143]
 [ 0.          0.          0.          0.         -3.        ]]
on enleve    la 2  me  ligne inf fois la  3  me  ligne
[[ 1.  3.  2. -5.  1.]
 [ 0. -7. -4. 11. -3.]
 [nan nan nan nan inf]
 [ 0.  0.  0.  0. -3.]]
on enleve    la 1  me  ligne inf fois la  3  me  ligne
[[ 1.  3.  2. -5.  1.]
 [nan nan nan nan inf]
 [nan nan nan nan inf]
 [ 0.  0.  0.  0. -3.]]
on enleve    la 0  me  ligne -inf fois la  3  me  ligne
[[ nan  nan  nan  nan -inf]
 [ nan  nan  nan  nan  inf]
 [ nan  nan  nan  nan  inf]
 [  0.   0.   0.   0.  -3.]]
on enleve    la 1  me  ligne nan fois la  2  me  ligne
[[ nan  nan  nan  nan -inf]
 [ nan  nan  nan  nan  nan]
 [ nan  nan  nan  nan  inf]
 [  0.   0.   0.   0.  -3.]]
on enleve    la 0  me  ligne nan fois la  2  me  ligne
[[nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan inf]
 [ 0.  0.  0.  0. -3.]]
```

```
# ON TERMINE

on enleve    la 0  me  ligne nan fois la  1  me  ligne
[[nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan inf]
 [ 0.  0.  0.  0. -3.]]
[[nan nan inf -3.]]
<ipython-input-19-620e4523503b>:21: RuntimeWarning: divide by zero
    encountered in double_scalars
  c = D[i,j]/D[j,j]
<ipython-input-2-263c5dffaf1e>:2: RuntimeWarning: invalid value
    encountered in multiply
  A[i,:] = A[i,:] + x*A[j,:]
<ipython-input-19-620e4523503b>:29: RuntimeWarning: divide by zero
    encountered in double_scalars
  Sol = dilatcolonne(NewB1,s,1./D[s,s])
Out[25]: array([[ nan,  nan,  nan, -inf]])
```

Donc cela ne marche pas !

```
# EXERCICE 06 CORRECTION
In [47]: A = np.array
    ([[1.,3.,2.,-5.],[1.,-4.,-2.,6.],[3.,1.,-2.,1.],[-2.,1.,-4.,3.]])
In [48]:  B = np.array([[1.],[-2.],[3.],[2.]])

In [44]: NEW_systLin(A,B)
[[ 1.  3.  2. -5.  1.]
 [ 1. -4. -2.  6. -2.]
 [ 3.  1. -2.  1.  3.]
 [-2.  1. -4.  3.  2.]]
On transforme en systeme triang sup :
On passe a la colonne C 0
indice ligne du coeff le plus grand: 2
[[ 3.  1. -2.  1.  3.]
 [ 1. -4. -2.  6. -2.]
 [ 1.  3.  2. -5.  1.]
 [-2.  1. -4.  3.  2.]]
le pivot est : 3.0
on enleve a la 1 eme ligne 0.3333333333333333 fois la 0 eme ligne
[[ 3.          1.         -2.          1.          3.        ]
 [ 0.         -4.33333333 -1.33333333  5.66666667 -3.        ]
 [ 1.          3.          2.         -5.          1.        ]
 [-2.          1.         -4.          3.          2.        ]]
```

```
ON CONTINUE

le pivot est : 3.0
on enleve a la 2 eme ligne 0.3333333333333333 fois la 0 eme ligne
[[ 3.          1.         -2.          1.          3.        ]
 [ 0.         -4.33333333 -1.33333333  5.66666667 -3.        ]
 [ 0.          2.66666667  2.66666667 -5.33333333  0.        ]
 [-2.          1.         -4.          3.          2.        ]]
le pivot est : 3.0
on enleve a la 3 eme ligne -0.6666666666666666 fois la 0 eme ligne
[[ 3.          1.         -2.          1.          3.        ]
 [ 0.         -4.33333333 -1.33333333  5.66666667 -3.        ]
 [ 0.          2.66666667  2.66666667 -5.33333333  0.        ]
 [ 0.          1.66666667 -5.33333333  3.66666667  4.        ]]
On passe a la colonne C 1
indice ligne du coeff le plus grand: 1
[[ 3.          1.         -2.          1.          3.        ]
 [ 0.         -4.33333333 -1.33333333  5.66666667 -3.        ]
 [ 0.          2.66666667  2.66666667 -5.33333333  0.        ]
 [ 0.          1.66666667 -5.33333333  3.66666667  4.        ]]
le pivot est : -4.333333333333333
on enleve a la 2 eme ligne -0.6153846153846154 fois la 1 eme ligne
[[ 3.          1.         -2.          1.          3.        ]
 [ 0.         -4.33333333 -1.33333333  5.66666667 -3.        ]
 [ 0.          0.          1.84615385 -1.84615385 -1.84615385]
 [ 0.          1.66666667 -5.33333333  3.66666667  4.        ]]
le pivot est : -4.333333333333333
on enleve a la 3 eme ligne -0.3846153846153846 fois la 1 eme ligne
[[ 3.00000000e+00  1.00000000e+00 -2.00000000e+00  1.00000000e+00
   3.00000000e+00]
 [ 0.00000000e+00 -4.33333333e+00 -1.33333333e+00  5.66666667e+00
  -3.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  1.84615385e+00 -1.84615385e+00
  -1.84615385e+00]
 [ 0.00000000e+00  2.22044605e-16 -5.84615385e+00  5.84615385e+00
   2.84615385e+00]]
On passe a la colonne C 2
indice ligne du coeff le plus grand: 3
[[ 3.00000000e+00  1.00000000e+00 -2.00000000e+00  1.00000000e+00
   3.00000000e+00]
 [ 0.00000000e+00 -4.33333333e+00 -1.33333333e+00  5.66666667e+00
  -3.00000000e+00]
 [ 0.00000000e+00  2.22044605e-16 -5.84615385e+00  5.84615385e+00
   2.84615385e+00]
 [ 0.00000000e+00  0.00000000e+00  1.84615385e+00 -1.84615385e+00
  -1.84615385e+00]]
le pivot est : -5.846153846153846
```

```
on enleve a la 3 eme ligne -0.3157894736842105 fois la 2 eme ligne
[[ 3.00000000e+00  1.00000000e+00 -2.00000000e+00  1.00000000e+00
   3.00000000e+00]
 [ 0.00000000e+00 -4.33333333e+00 -1.33333333e+00  5.66666667e+00
  -3.00000000e+00]
 [ 0.00000000e+00  2.22044605e-16 -5.84615385e+00  5.84615385e+00
   2.84615385e+00]
 [ 0.00000000e+00  7.01193489e-17 -2.22044605e-16  6.66133815e-16
  -9.47368421e-01]]

On transforme en systeme diagonal :

on enleve a la 2 eme ligne 8776245427696351.0 fois la 3 eme ligne
[[ 3.00000000e+00  1.00000000e+00 -2.00000000e+00  1.00000000e+00
   3.00000000e+00]
 [ 0.00000000e+00 -4.33333333e+00 -1.33333333e+00  5.66666667e+00
  -3.00000000e+00]
 [ 0.00000000e+00 -6.15384615e-01 -3.89743590e+00  0.00000000e+00
   8.31433777e+15]
 [ 0.00000000e+00  7.01193489e-17 -2.22044605e-16  6.66133815e-16
  -9.47368421e-01]]
on enleve a la 1 eme ligne 8506799296144271.0 fois la 3 eme ligne
[[ 3.00000000e+00  1.00000000e+00 -2.00000000e+00  1.00000000e+00
   3.00000000e+00]
 [ 0.00000000e+00 -4.92982456e+00  5.55555556e-01  0.00000000e+00
   8.05907302e+15]
 [ 0.00000000e+00 -6.15384615e-01 -3.89743590e+00  0.00000000e+00
   8.31433777e+15]
 [ 0.00000000e+00  7.01193489e-17 -2.22044605e-16  6.66133815e-16
  -9.47368421e-01]]
on enleve a la 0 eme ligne 1501199875790165.2 fois la 3 eme ligne
[[ 3.00000000e+00  8.94736842e-01 -1.66666667e+00  0.00000000e+00
   1.42218936e+15]
 [ 0.00000000e+00 -4.92982456e+00  5.55555556e-01  0.00000000e+00
   8.05907302e+15]
 [ 0.00000000e+00 -6.15384615e-01 -3.89743590e+00  0.00000000e+00
   8.31433777e+15]
 [ 0.00000000e+00  7.01193489e-17 -2.22044605e-16  6.66133815e-16
  -9.47368421e-01]]
```

```
on enleve a la 1 eme ligne -0.14254385964912283 fois la 2 eme ligne
[[ 3.00000000e+00  8.94736842e-01 -1.66666667e+00  0.00000000e+00
   1.42218936e+15]
 [ 0.00000000e+00 -5.01754386e+00  0.00000000e+00  0.00000000e+00
   9.24423081e+15]
 [ 0.00000000e+00 -6.15384615e-01 -3.89743590e+00  0.00000000e+00
   8.31433777e+15]
 [ 0.00000000e+00  7.01193489e-17 -2.22044605e-16  6.66133815e-16
  -9.47368421e-01]]
on enleve a la 0 eme ligne 0.4276315789473685 fois la 2 eme ligne
[[ 3.00000000e+00  1.15789474e+00  0.00000000e+00  0.00000000e+00
  -2.13328403e+15]
 [ 0.00000000e+00 -5.01754386e+00  0.00000000e+00  0.00000000e+00
   9.24423081e+15]
 [ 0.00000000e+00 -6.15384615e-01 -3.89743590e+00  0.00000000e+00
   8.31433777e+15]
 [ 0.00000000e+00  7.01193489e-17 -2.22044605e-16  6.66133815e-16
  -9.47368421e-01]]
on enleve a la 0 eme ligne -0.2307692307692308 fois la 1 eme ligne
[[ 3.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   1.25000000e+00]
 [ 0.00000000e+00 -5.01754386e+00  0.00000000e+00  0.00000000e+00
   9.24423081e+15]
 [ 0.00000000e+00 -6.15384615e-01 -3.89743590e+00  0.00000000e+00
   8.31433777e+15]
 [ 0.00000000e+00  7.01193489e-17 -2.22044605e-16  6.66133815e-16
  -9.47368421e-01]]
[[ 1.25000000e+00  9.24423081e+15  8.31433777e+15 -9.47368421e-01]]
Out[44]:
array([[ 4.16666667e-01, -1.84238167e+15, -2.13328403e+15,
        -1.42218936e+15]])
```

On peut interpréter ces solutions très grandes. En effet si l'on tape **`alg.det(A)`**, on obtient **`2.6850316841703656e-14`** qui est très proche de 0 et donc le système est quasi non inversible.