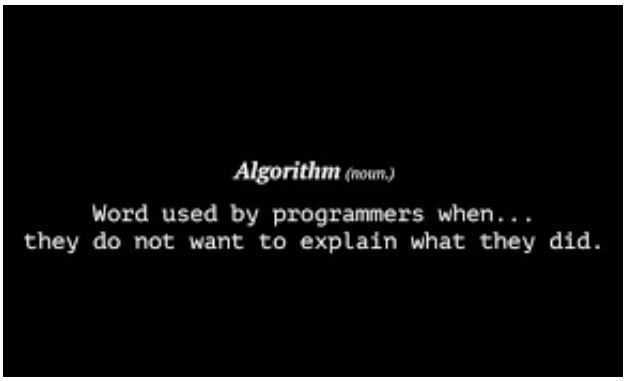
Chapitre 1
Algorithmie

Simon Dauguet
simon.dauguet@gmail.com

30 septembre 2024



Nous allons ici entrer dans le cœur du sujet. On va ici introduire l'algorithmie et les règles qui la régissent. On se placera dans le cadre particulier du langage Python et tout sera présenté selon ce point de vue. Dans un autre langage, il faudrait tout modifier pour respecter les règles syntaxiques du nouveau langage.

Table des matières

1 Le langage Python

1.1 Langage informatique

Définition 1.1 (Langage informatique, Algorithme) :

- Un langage informatique est un langage permettant de demander à un ordinateur d'effectuer des tâches, des calculs.
- On appelle algorithme une tâche donnée à faire à l'ordinateur. Un algorithme est donc une suite d'ordres élémentaires que peut effecteur l'ordinateur.

Pour pouvoir donner ces ordres à la machine, il est nécessaire de communiquer avec elle. Le moyen pour le faire est ce qu'on appelle un langage informatique. Il existe une multitude de langage informatique. Le principe de base est toujours le même. Un algorithme est toujours construit selon les mêmes règles. Les commandes sont donc essentiellement les mêmes. Les différences entre les différents langages sont plutôt de l'ordre syntaxique.

Par exemple, en langage Maple, toute tâche doit obligatoirement se terminer par un point-virgule. C'est ce qui permet à l'ordinateur de reconnaître la fin d'un ordre. Il y en a d'autres qui nécessitent de mettre chaque ordre entre accolades.

Les mots-clés qui permettent de faire un algorithme changent également légèrement d'un langage à l'autre. Mais les différences sont essentiellement esthétiques.

Pour effectuer un algorithme, il faudra donc procéder en 3 étapes :

(i) Écrire (ou coder) l'algorithme (en faisant attention aux règles syntaxiques du langage utilisé).

(ii) Compiler le code. C'est l'étape où l'ordinateur lit la chaîne d'instructions de l'algorithme pour savoir ce qu'il a à faire (et voir aussi s'il n'y a pas d'erreur de syntaxe). C'est l'étape la plus primordiale et celle qui cause le plus de soucis.

(iii) Utilisation de l'algorithme. Une fois compilé, l'algorithme est dans la mémoire de l'ordinateur, mais il n'a pas encore été utilisé. À cette étape, il est juste créé. Il reste à en faire quelque chose.

La première étape débutera dans le prochain cours où nous verrons les règles de l'algorithmique (dont vous avez déjà eu quelques notions dans les petites classes). Dans ce chapitre, on va surtout s'attarder sur le dernier point : comment parler à l'ordinateur directement et lui donner quelques ordres immédiats.

1.2 Le Langage Python

Le langage Python a été créé au début des années 1990 par Guido van Rossum. En 2001, la PSF (Python Software Foundation) est créée. C'est une organisation à but non lucratif propriétaire des droits intellectuels de Python qui distribue Python sous forme de logiciels libres. Il existe plusieurs plateformes de codage en Python. Nous utiliserons Spyder3. Mais lors des oraux, vous utiliserez IDLE. Les différentes interfaces fournissent différents trucs et astuces qui permettent de se simplifier un peu la vie. Il est conseillé, de temps en temps, de passer d'une interface à l'autre pour bien être conscient de ce qui fait partie du langage Python de ce qui fait partie des fonctionnalités de l'interface.

Les langages informatiques sont classés en différentes catégories selon les possibilités qu'ils offrent. En ce qui concerne Python, c'est un langage interprété, orienté objet, de haut niveau, modulaire, à syntaxe positionnelle, au typage dynamique.

- **Langage interprété :** Python est dit interprété car il est directement exécuté sans passer par une phase de compilation qui traduit le programme en langage machine, comme c'est le cas pour le langage C. En quelque sorte, il fonctionne autant comme une calculatrice que comme un langage de programmation. Afin d'accélérer l'exécution d'un programme Python, il est traduit dans un langage intermédiaire qui est ensuite interprété par une machine virtuelle Python. Ce mécanisme est semblable à celui propre au langage Java.
- **Langage orienté objet :** Python intègre le concept de classe ou d'objet. Un objet regroupe un ensemble de données et un ensemble de fonctionnalités attachées à ces données. Ce concept relie la description des données et les algorithmes qui leur sont appliqués comme un tout indissociable. Un objet est en quelque sorte une entité autonome avec laquelle on peut communiquer via une interface.
- **Langage de haut niveau :** il dispose de fonctionnalités avancées et automatiques de simplification d'écriture d'un code. Comme par exemple le *garbage collecting* qui détruit ce qui n'est plus utilisé (variables, fonctions ...).
- **Langage modulaire :** le noyau de fonctionnement de Python est très succinct et toutes les fonctionnalités annexes qu'offre Python sont contenues dans des modules complémentaires (aussi appelés bibliothèques ou paquets), ce qui le rend très polyvalent.
- **Syntaxe positionnelle :** Chaque langage informatique a ses propres règles de syntaxe, d'écriture. La syntaxe sert essentiellement à délimiter les commandes. C'est une sorte de ponctuation qui permet à l'ordinateur de délimiter le début d'un ordre et la fin de celui-ci. Le langage Python fonctionne sur la position. C'est-à-dire que c'est la position, l'indentation des lignes de code qui détermine les différentes strates du code. Un bloc d'instruction doit être au même niveau d'indentation. Cette syntaxe a l'avantage de faciliter la lecture d'un code. Et l'indentation est de toute façon utilisée dans tous les codes pour des questions de lisibilité, mais est sans incidence sur le bon fonctionnement du code avec les autres langage.
- **Typage dynamique :** Le type d'une variable est défini lors de l'exécution du programme et non lors de la compilation. La validité de telle ou telle opération est donc vérifiée au moment où cette opération est effectivement faite et pas lors de la transcription du code en langage machine (autrement dit, pas besoin de définir le type d'une variable. Le type est déterminé automatiquement en fonction de ce que l'on fait à cette variable).

1.3 Variables et Mots réservés

Un algorithme étant un programme de calcul, une suite d'instructions réalisant plusieurs tâches, il dépend le plus souvent de paramètres. C'est une version informatisée des fonctions mathématiques, en quelque sorte. Il faut donc utiliser des variables.

Une variable est un nom pouvant être une suite de lettres minuscules ou majuscules et de chiffres. Le nom de la variable doit commencer toutefois obligatoirement par une lettre. Et les lettres doivent être au naturel : pas d'accent, de cédilles etc.

Pour clarifier la lecture des algorithmes, il est fortement conseillé de donner des noms aux variables qui permettent de décrire également leur rôle. Par exemple, si l'on a besoin de deux variables pour

faire un jeu de transfert de l'une à l'autre, on pourrait appeler la première **`NewValeur`** et la seconde **`OldValeur`**, ou bien **`valeur1`** et **`valeur2`**.

Attention ! La casse est significative ! Changer une lettre minuscule par sa majuscule ou l'inverse change le nom de la variable ce qui entraînera en une erreur : Python ne reconnaîtra pas le nom de la variable et ne saura pas quoi en faire !

Par exemple, Joe, joe, JOE, JOe, JoE, jOE, jOe sont des noms de variables toutes différentes.

Attention toutefois, il y a certains mots qui sont réservés. Ce sont des mots clés pour le système qui ont un sens particulier. Ce sont en fait les mots-clés permettant de coder. En Python, les principaux mots-clés sont :

```
1 and       as      assert     break   class
2 continue  def     del        elif    else
3 except    exec    finally    for     from
4 global    if      import     in      is
5 lambda    not     or         pass    print
6 raise     return  try        while   ...
```

On ne pourra donc pas donner ces mots comme nom à une variable. Il sera facile de s'en rendre compte : Spyder repère automatiquement les mots-clés et les colorise automatiquement pour les faire ressortir.

Remarque (Commentaire) :
Il est fortement conseillé de commenter un script. Ça permet de pouvoir comprendre plus facilement ce que l'on a voulu faire si une ligne est fausse. Ça aide à la lecture d'un algorithme. Les commentaires se font grâce au symbole dièse : tout ce qui est après le symbole "#" dans une ligne est grisé et considéré comme un commentaire.

1.4 Opérations de bases

Définition 1.2 (Opérations de bases) :
Les opérations de bases sont les suivantes :

Commande	Définition	Exemple
`+` `-`	Addition, Soustraction	`x = y + z`
`*` `/`	Multiplication, Division	`x = y*z`
`**`	Puissance (entière ou réelle)	`x = y**0.5`
`//`	Quotient d'une division euclidienne. Le résultat est flottant si l'un des nombres est flottant.	`x = y//3`
`%`	Reste de la division euclidienne (cf congruence)	`x = y%3`

Il existe deux autres opérations que l'on peut souligner. Elles sont surtout utiles au sein d'un algorithme. Mais la notation est un peu contre-intuitive. On peut tout à fait s'en passer (ce que je fais personnellement).

Commande	Définition	Exemple	Équivalent
`+=`	Incrémentation de la variable	`x+=3`	`x = x+3`
`-=`	Décrémentation	`x-=3`	`x = x-3`
`*=`	Multiplication et affectation	`x*=3`	`x = x*3`
`/=`	Division et affectation	`x/=3`	`x = x/3`

Définition 1.3 (`print()`) :
La fonction `print(nom)` permet l'affichage de `nom`. Ce n'est qu'un affichage. Il n'a pas de type. On ne peut rien en faire.

L'intérêt de la fonction `print` est essentiellement de permettre un affichage plus lisible. Ce n'est que pour faciliter la vie de l'être humain qui tape sur le clavier parce qu'il ne sait pas lire correctement le langage informatique. Ce n'est que pour faire joli pour nous et détruit toutes les informations de ce qu'on affiche.

Cependant, la fonction `print` est très pratique pour déboguer un algorithme. Il permet de pouvoir voir ce qu'il se passe dans l'algorithme, pas à pas, sans freiner son exécution.

Exemple 1.1 :
Commenter.

```
>>> r,pi = 12,3.14159
>>> s=pi*r**2
>>> print(s)
452.38896
>>> print(type(s),type(r),type(pi))
<class 'float'> <class 'int'> <class 'float'>
>>> msg="Aire du disque"
>>> msg
"Aire du disque"
>>> print(msg)
Aire du disque
```

Remarque :
Il est parfaitement possible de composer les différentes fonctions de Python. Par exemple, on peut très bien faire `print(7+3)` ou encore `print(type(7**2+2**3.1))`. Il faudra simplement faire attention aux nombres de parenthèses...

1.5 Manuel d'aide

Il est régulier d'oublier comment on utilise une commande. Et c'est là que la commande d'aide rentre en action. Il suffit de taper `help(nom)` pour avoir une aide sur `nom`. Si c'est une fonction, on aura sa syntaxe et ce à quoi elle sert. La fonction `help()` fonctionne avec à peu près n'importe quoi.

Il va falloir apprendre à lire ces manuels. Le but de ces manuels est de donner toutes les informations pour pouvoir utiliser les commandes dans toutes les situations, y compris les compliquées qu'on ne rencontrera pas. Il faudra donc faire le tri.

2 Les types en python

Python (et de façon plus générale, chaque langage informatique) fait très attention aux types des objets qu'il doit manipuler. Manipuler un entier, ce n'est pas pareil que manipuler un réel. Et comme un réel n'est pas un entier, Python ne va pas traiter ces deux types de la même manière.

Chaque type d'objet a ses opérations propres. Et on ne pourra mélanger les opérations entre les types sous peine de tomber sur une erreur. Il y a donc une addition pour les entier, une addition pour les réels, une multiplication pour les entiers, une multiplication pour les réels etc.

Python manipule principalement 3 types d'objets (on en verra d'autres par la suite) :

Définition 2.1 (Types de bases) :

- `int` : de l'anglais *integer*. Ce sont les entiers. Positifs ou négatifs. Ce sont des nombres sans décimales.
- `float` : littéralement, flottant. Plus exactement, les décimaux. Python étant limité phy-

siquement, il ne peut pas manipuler un nombre infini de décimales et donc tronque les décimales des réels. Il va y avoir des problèmes d'arrondis, entre autres choses, qui vont apparaître. Les flottants ont TOUJOURS au moins une décimale. Même si c'est 0. Et ce ne sont pas des entiers.

- `str` : de l'anglais *string*. Littéralement chaîne. Il faut comprendre chaîne de caractère. C'est une suite de caractères. L'espace étant un caractère particulier.
- `list` : listes. Le nom est plutôt clair. C'est une liste de différents machins. Chacun des éléments de la liste n'ont pas obligatoirement le même type. On réétudiera les listes et les chaînes de caractères plus en détail dans le chapitre 4.
- `bool` : booléen. Ce type correspond aux (deux) objets logiques (voir maths, chap Logique).

Il n'est pas nécessaire de prédéfinir le type d'une variable avant de lui donner une valeur. La variable prend le type de la valeur qu'on lui affecte. Dès qu'on réaffecte une variable, son type change de nouveau pour prendre le même que celui de la valeur qu'on lui a affectée.

Définition 2.2 (`type()`) :
La commande `type(variable)` permet d'avoir le type de la variable `variable`. C'est ce qui est après le point qui donne le type de la variable.

Exemple 2.1 :
Décrire le plus clairement possible ce qu'il se passe avec les lignes de l'interpréteur suivantes :

```
1 >>> largeur=20
2 >>> hauteur=5*9.3
3 >>> largeur*hauteur
4 930.0
```

2.1 Booléen

Définition 2.3 (Booléen) :
Les booléens ont un rôle très important en algorithmie. Ce sont des résultats d'opérations logiques simples qui ne peuvent prendre que deux valeurs possibles : `True` ou `False`.

Ce sont donc des réponses à des questions de vérification. Soit l'énoncé est vrai, soit il est faux.

Définition 2.4 (Opérateurs de comparaison) :
Il existe 3 types d'opérateurs de comparaison :

Opérateurs	Définition	Syntaxe
`<` `>`	Inférieur strict, Supérieur strict	`x<5`
`<=` `>=`	Inférieur, supérieur (large)	`x<=5`
`==` `!=`	Égal, Différent	`x==5` ou `x!=5`

Ces opérateurs s'appliquent à tous les types. Évidemment pour avoir une chance d'obtenir une réponse vraie, il faut que les types comparés soient identiques.

Exemple 2.2 :
Commenter :

```
>>> 5=="5"
False
>>> 5!="5"
True
>>> 7-3<5
True
>>> (type(12**12-17**17)==type(5**0.2)) == ((2.5**15)>(2.1**7.8))
False
```

Remarque :
Il faut user et abuser des parenthèses en algorithmie pour que l'ordinateur comprenne bien quel groupe doit être en lien avec quel groupe. Il suffit de reprendre la dernière ligne et enlever une paire de parenthèses pour tomber sur une erreur.

Les erreurs de parenthèses sont des erreurs classiques qui arrivent très souvent. Plus vous y penserez, mieux ce sera. Et il faut savoir aussi repérer une erreur de parenthèses.

On peut faire des opérations entre booléens aussi :

Définition 2.5 (Opérateurs booléens) :
On recense essentiellement 3 opérateurs booléens :

Opérateur	Définition	Syntaxe
`and`	Et (conjonction)	`x=((5<6) and (5==0))`
`or`	Ou (disjonction)	`y=((5<6) or (5==0))`
`not`	Négation	`x==(not y)`

On peut également mettre dans les opérateurs `is` et `in` qui sont respectivement l'opérateur d'identification et d'appartenance. Ils sont assez intuitifs dans leur utilisation, mais on les reverra en temps utile (surtout le second).

Exemple 2.3 :
Commenter

```
>>> x = (5-6<7*2-8**2)
>>> y = (7*8-5**2+6 >= 9*13-7*17)
>>> (x and y) or (x or y)
True
>>> ((x or (not y)) or (not x)) and (y or not(x and y)) and (not x)
True
```

2.2 Listes

2.2.1 Présentation générale

Définition 2.6 (Liste) :
Une liste est un ensemble de données ordonnées. Les données peuvent être de même nature (liste homogène) ou non (liste hétérogène). Pour une liste contenant n valeurs, le premier est le numéro 0 et le dernier est le $n-1$. Une liste est délimitée par des crochets et les différents éléments sont séparés par des virgules.

Exemple 2.4 :

```
>>> L=[27,1.2,"bla"]
```

est une liste hétérogène composée de 3 éléments. Le premier est l'entier 27 et le dernier est la chaîne `"bla"`.

Remarque :
Une liste est donc un ensemble de données ordonnées, mais pas selon la valeur des éléments qui la composent. Il y a simplement un premier élément, puis un second etc. Les éléments sont numérotés et ils sont ordonnés selon leur numéro.

Définition 2.7 (Numéro d'index) :
Pour une liste `L` contenant n éléments, l'index est le numéro de chaque élément de la liste. L'index est donc un entier allant de 0 à $n-1$.
Si $i \in [\![0, n-1]\!]$, l'élément d'index i de la liste `L` est obtenu avec la commande `L[i]`.

!!! ATTENTION !!!

⚠ Le premier élément d'une liste est le numéro 0 ! Python commence toujours à compter à partir de 0. Ce qui engendre certains problèmes de comptage. Il ne faut pas l'oublier. C'est une erreur fréquente qui aboutit en général à une erreur `IndexError: list index out of range`.

!!! ATTENTION !!!

⚠ Une fonction s'utilise avec des parenthèses, une liste avec des crochets ! Il ne faut pas intervertir les deux, sans quoi on tombe sur une erreur système du type `TypeError: 'list' object is not callable`.

2.2.2 Création d'une liste

Il y a plusieurs façons de créer une liste. Les plus simples et plus courantes sont les suivantes :

- Définition explicite : On créée la liste L telle qu'elle est. Par exemple `L=[a,b,c]` est une liste de trois éléments. On peut également définir une liste pour pouvoir plus facilement ajouter des éléments par la suite sans gêne : `L=[]`.

- Définition en intention : On créée une liste à partir de l'expression d'une suite. `L=[expression for k in range(n,m) condition]`. Cela correspond mathématiquement à $L = \{f(k), k \in \{n, \dots, m-1\}\}$.

Exemple 2.5 :

```
>>> L=[0,1,2,5]
>>> L
[0,1,2,5]
>>> L=[k*(k+1)//2 for k in range(0,11) if k%2==0]
>>> L
[0, 3, 10, 21, 36, 55]
```

La dernière syntaxe est très pratique. Mais dans le cas où l'on veut simplement énumérer des éléments à la suite, on peut aussi utiliser :

Définition 2.8 (`list()`) :
La fonction `list` (sans e, en anglais) permet de fabriquer des listes facilement et rapidement à partir d'itérables. L'utilisation la plus courante est

```
>>> list(range(n,m))
```

qui donne la liste de tous les éléments entre n et $m-1$.

On peut voir la commande `list` comme une commande de transtypage qui transforme un itérable en liste.

Exemple 2.6 :

```
>>> list(range(0,10))
[0,1,2,3,4,5,6,7,8,9]
>>> list("abcde")
['a','b','c','d','e']
>>> list([1,2,3,4])
[1,2,3,4]
```

Remarque :
Attention, un `range` n'est pas une liste et ne s'utilise pas de la même manière. Le type du `range` est

un type particulier, à part entière. Il faut donc transformer un `range` en liste pour pouvoir l'utiliser en tant que telle, grâce à la fonction `list`. Néanmoins, il y a certaine utilisation des range qui se font de la même manière que les listes : `len(range(n,m))` renvoie $m-n$; `range(n,m)[i]` renvoie le $(i-1)$-ème entier entre n et m, c'est-à-dire l'entier $n+i$.

2.2.3 Utilisation des index

L'index d'une liste `L` est automatiquement fourni avec la liste lors de la définition de la liste. L'index est attaché à sa liste. L'un ne va pas sans l'autre. C'est un outil assez flexible qui permet (entre autres choses) d'avoir accès à différentes informations sur la liste :

Commandes de manipulations d'index dans une liste

Commandes	Description
`len(L)`	(*length*) Nombre d'éléments de la liste. C'est donc aussi l'index du dernier élément de la liste $+1$.
`L[i]`	Élément d'index i de la liste `L`. Ce qui correspond donc au $(i+1)$-ème élément. Ce n'est pas une liste. C'est vraiment l'élément de la liste. Ce qui permet de pouvoir le modifier.
`L[i:j]`	Sous-liste de tous les éléments compris entre les index i et $j-1$.
`L[i:j:k]`	Sous liste de tous les éléments de `L` compris entre les index i et $j-1$ avec un pas de k.
`L[::k]`	Correspond à `L[0:n:k]`. C'est donc tous les éléments de la liste en comptant de k en k en partant du premier élément.
`L[-1]`	Dernier élément de la liste (quel que soit le nombre d'élément de la liste). Ce qui correspond donc aussi à `L[len(L)-1]`. De façon similaire, on a aussi `L[-2]` qui renvoie l'avant-dernier élément de la liste ; `L[-3]` qui renvoie l'avant-avant-dernier élément de la liste etc. On peut donc compter à l'envers.

Exemple 2.7 :

```
>>> L=[7,8,4,2,9,6,15,3,487,26,24]
>>> len(L)
11
>>> L[0]
7
>>> L[11]
IndexError : list index out of range
>>> L[10]
24
>>> L[1:10]
[8,4,2,9,6,15,3,487,26]
>>> L[1:10:3]
[8,9,3]
>>> L[5]="changement d'element"
>>> L
[7, 8, 4, 2, 9, "changement d'element", 15, 3, 487, 26, 24]
>>> L[-1],L[-2],L[-3]
24, 26, 487
>>> L[-len(L)], L[len(L)-1]
7, 24
```

2.2.4 Opérations sur les listes

Définition 2.9 (Concaténation) :
Dans certains langages de programmation, enchaînement de deux listes ou de deux chaînes de caractères mises bout à bout. (*Larousse*)

Définition 2.10 (Opérations + et * pour les listes) :

- La concaténation de deux listes se fait avec l'opération $+$. On ne peut additionner que des listes entres elles. Sinon, on a une erreur système du type `TypeError`. Attention à l'ordre dans lequel on écrit les listes. La concaténation n'est pas commutative.
- La répétition d'une liste se fait avec *. La syntaxe est `L*k` où k est un entier et correspond à `L+L+...+L` où la concaténation se fait k fois. On peut écrire aussi `k*L`.

Globalement, les opérations sont les mêmes que pour les chaînes de caractères. On peut voir les chaînes de caractères comme des sortes de listes particulières.
Exemple 2.8 :

```
>>> L=[2,5,78,3]
>>> LL=[7,0,-5]
>>> L+LL
[2, 5, 78, 3, 7, 0, -5]
>>> LL+L
[7, 0, -5, 2, 5, 78, 3]
>>> LL*2
[7, 0, -5, 7, 0, -5]
>>> 3*LL
[7, 0, -5, 7, 0, -5, 7, 0, -5]
```

Il existe cependant beaucoup d'autres fonctions qui permettent de pouvoir modifier les listes. On peut citer par exemple :

Fonctions de modification d'une liste

Commande	Description
`NomListe.append(x)`	Ajoute l'élément `x` à la fin de la liste. Cette opération correspond donc à `NomListe=NomListe+[x]`.
`NomListe.pop(k)`	Supprime et retourne l'élément d'index k.
`del NomListe[i]`	Supprime l'élément d'index i de la liste et ne renvoie rien. On peut supprimer aussi plusieurs éléments d'un coup avec `del NomListe[i:j:k]`.
`NomListe.remove(x)`	Si x est un élément de la liste, supprime la première occurrence de l'élément x de la liste (mais ne renvoie rien). Si x n'est pas un élément de la liste, renvoie un message d'erreur.
`NomListe.insert(k,x)`	Insère l'élément x à la place d'index k. La taille de la liste est donc augmentée de 1. Et les index des éléments après le k-ème sont augmentés aussi de 1 (ils sont donc décalés).
`NomListe.reverse()`	Renverse les éléments de la liste. C'est un comptage à rebours sur les index.
`min(NomListe)`	Renvoie l'élément de la liste avec la plus petite valeur dans l'ordre lexicographique si la liste est homogène. Il ne faut que des éléments comparables dans la liste.
`max(NomListe)`	Idem que `min` mais avec le maximum.

`NomListe.count(val)` Renvoie le nombre d'occurrences de la valeur `val` dans la liste.

Exemple 2.9 :

```
>>> Liste=[2,7,-2,5,2]
>>> min(L)
-2
>>> Liste.append("dernier element")
>>> Liste
[2,7,-2,5,2,"dernier element"]
>>> Liste.pop(-1)
"dernier element"
>>> Liste
[2,7,-2,5,2]
>>> Liste.remove(2)
>>> Liste
[7,-2,5,2]
>>> del L[-1]
>>> Liste
[7, -2, 5]
>>> Liste.insert(1,"insertion")
>>> Liste
[7, 'insertion', -2, 5]
>>> Liste.reverse()
>>> Liste
[5, -2, 'insertion', 7]
>>> max(Liste)
TypeError: unorderable types: str() > int()
>>> Liste[2]=5
>>> Liste.count(5)
2
```

2.2.5 Copies de Listes

Il peut être utile de faire une copie d'une liste pour pouvoir la manipuler sans la modifier. Si `L` est une liste, faire `LL=L` ne suffira pas. En effet, la nouvelle liste `LL` SERA la liste `L`. Ce ne sera pas une copie. Donc modifier `LL` modifiera également la liste initiale.

Pour pouvoir faire une copie, il faut vraiment faire une copie, c'est à dire créer une liste dont les éléments sont les mêmes que `L`. En d'autres termes, il faut copier les éléments (un par un) de `L`.

Exemple 2.10 :

```
>>> L=[0,1,2,3,4]
>>> LL=L
>>> Lbis=L[:]
>>> Lter=[L[k] for k in range(0,len(L)]
>>> L==LL, L==Lbis, L==Lter
(True, True, True)
>>> LL[0]="changement"
>>> L,LL
(['changement', 1, 2, 3, 4], ['changement', 1, 2, 3, 4])
>>> Lbis[0]="autre"
>>> Lbis
['autre', 1, 2, 3, 4]
>>> Lter[0]="different"
>>> Lter
['different', 1, 2, 3, 4]
>>> L,Lbis
(['changement', 1, 2, 3, 4], ['autre', 1, 2, 3, 4])
>>> L,Lter
(['changement', 1, 2, 3, 4], ['different', 1, 2, 3, 4])
```

2.3 Chaînes de caractères

Définition 2.11 (Chaîne de caractères) :
Une chaîne de caractères (ou *string* en anglais) est une suite finie de caractères, une suite de symboles. C'est donc du texte avec tout ce que ça signifie. La casse est importante.

Le début et la fin d'une chaîne de caractères est marqué par des guillemets, des apostrophes, ou trois guillemets à la suite ou trois apostrophes à la suite. Ils ne servent qu'à délimiter le début et la fin de la chaîne. Ils ne font pas partie de la définition de la chaîne.

!!! ATTENTION !!!

⚠ Un ordinateur ne sait pas lire ! Même si ce n'est pas toujours évident, il faut donc se défaire de nos mauvaises habitudes qui consiste à lire et comprendre un texte que l'on voit. Un texte, ce n'est qu'une suite de symboles. Et c'est tout. C'est en ce sens que l'ordinateur le comprend. Un seul symbole qui est changé modifie donc la chaîne de caractères.

Exemple 2.11 :
Commenter :

```
>>> MSG='Hello World !'
>>> mSG="hello World !"
>>> Msg="""Hello World !"""
>>> MSG==Msg
True
>>> Msg==msg
False
>>> ""Hello World""
SyntaxError : invalid syntax
```

Remarque :
Attention, c'est du texte simple. Pas d'accent ni aucune fantaisie. Juste l'alphabet de base en minuscule ou majuscule et la ponctuation.

Il peut être utile aussi de mettre en forme un texte. Par exemple, on pourrait vouloir un texte composé de deux mots, chacun sur une ligne. Il faut des symboles particuliers pour cela :

Commande	Définition	Syntaxe
\'	Apostrophe	"L\'homme"
\"	Guillemet	"Il dit : \"Ce n'est pas si complique !\""
\\	Backslash	"Du sucre et\\ou du lait ?"
\n	Nouvelle ligne	"La sanglots longs\nDes violons\nDe l'automne"
\%	Pourcentage	"Il a eu une reduction de 3.5\%"
\t	Tabulation	"Une colonne\tUne autre colonne\n1\t2"

Remarque :
Certains de ces caractères spéciaux sont indirectement utiles. C'est le cas de \' et \". Même si on ne les voit pas toujours, Python les inscrit automatiquement mais l'affichage ne les fait pas forcément apparaître. Pour plus de clarté. Néanmoins, ça devient utile lors du changement d'OS. Dans le passage de Windows à Linux on Mac qui n'utilise pas tout à fait les mêmes codages, on peut se retrouver avec quelques mauvaises surprises.

D'une façon générale, les trois guillemets ou les trois apostrophes servent à définir une chaîne longue qui peut s'étendre sur plusieurs lignes (En fait, ils ont une autre utilité et c'est pour ça qu'ils apparaissent en italique, mais c'est une autre histoire). On ne met jamais deux guillemets ou deux apostrophes pour tenir compte du fait que le texte peut en contenir et pour ainsi ne pas couper le texte en deux.

Les commandes de sauts de ligne et de tabulation s'utilisent à la place voulue sans espacement. Tout espacement supplémentaire serait considéré comme tel et apparaîtra dans le rendu final. Donc " \t " est une chaîne de caractères contenant 3 caractères mais dont l'espace blanc est le même

que celui d'une tabulation et de deux espaces. De même, `"\n "` est un changement de ligne avec un espace après le saut de ligne. Il y aura donc une indentation dans le texte.

La mise en forme d'un texte n'est clairement pas pratique en Python. Ce n'est pas un traitement de texte.

Une chaîne de caractères est une liste. On peut donc lui appliquer les mêmes opérations que les listes. Nous verrons les listes plus en détail un peu plus tard.

Définition 2.12 (Concaténation) :
Dans certains langages de programmation, enchaînement de deux listes ou de deux chaînes de caractères mises bout à bout. (Larousse)

Pour plus d'exemples sur la notion de concaténation, voir le cours de maths sur les dimensions finies.

Opérateur	Définition	Syntaxe
`+`	Concaténation de deux chaînes	`S="Il dit" + "non"`
`+=`	Concaténation et affectation	`S+="\nPuis il dit oui"`
`in`, `not in`	Inclusion (ou non inclusion)	`"oui" in S`
`*`	Répétition d'une chaîne de caractères (commutatif)	`T="abc"*4` ou `4*"abc"`
`chaine[n]`	Accession au $n$ème caractère de la chaîne	`"abc"[0]`
`chaine[n:m]`	Sous-chaîne contenue entre les caractères n et $m-1$	`4*"abc"[2:6]`
`len()`	Longueur de la chaîne (nombre de caractères)	`len(S)`

Définition 2.13 (`ord` et `chr`) :
La fonction `ord(let)` prend en argument une chaîne de caractère `let` composée d'un seul caractère et renvoie un entier correspondant au code de la lettre `let` en Unicode (voir le chapitre sur le codage plus tard dans l'année).

La lettre `"a"` correspond à l'entier 97. De sorte que toutes les lettres minuscules de l'alphabet seront "bien" numérotées en effectuant la commande `ord("a")-97`.

La fonction `chr(n)` prend en argument un entier `n` et renvoie la lettre correspondant dans le code Unicode. C'est le "contraire" de la fonction `ord`.

Exemple 2.12 :

```
>>> ord("a")
97
>>> ord("A")
65
>>> ord("z")-97
25
>>> chr(25+97)
"z"
>>> chr(66)
"B"
```

Définition 2.14 (Chaîne de caractère avec variable : Méthode `f`) :
Depuis `python3.6`, afin d'afficher une chaîne de caractères contenant des variables, on préférera utiliser la méthode `f`. Cette méthode permet de remplacer une variable par la traduction de sa valeur en tant que chaîne caractères.

Exemple 2.13 :

```
>>> age = 18
>>> message = f"Je suis agé de {age} ans."
>>> print(message)
Je suis agé de 18 ans.
>>> print("Je suis agé de",age,"ans.")
Je suis agé de 18 ans.
```

2.4 Conversion - Transtypage

Le problème est simple. Si l'on a un nombre dans une chaîne de caractère, c'est une chaîne et pas un nombre. On ne peut pas calculer avec. Il faut pouvoir récupérer la valeur contenue à l'intérieur de la chaîne.

Le problème fonctionne aussi dans l'autre sens (et c'est le plus fréquent). Mettons que l'on ait `x=5.2`. On veut maintenant mettre la valeur de la variable `x` dans une chaîne. Mais pas le nom de la variable. Donc taper `"x"` nous donne une chaîne de caractères contenant la lettre x. Et pas la valeur de la variable. Il faut donc convertir la variable en chaîne.

Commande	Définition	Syntaxe
`int()`	Convertit une chaîne de caractère ne contenant qu'un entier en cet entier	`>>> int("3")+2` `5`
`float()`	Convertit une chaîne de caractère ne contenant qu'un flottant ou un entier en flottant	`>>> float("3.2")+2` `5.2`
`str()`	Convertit un nombre ou une variable en une chaîne	`>>> str(3.6)` `"3.6"`

Bien sûr, on peut composer les fonctions : `str(3.14159+2.789)`.

Exemple 2.14 :
Commenter

```
1 >>> pi=3.14159
2 >>> Pi=str(pi)
3 >>> float(Pi)
4 3.14159
5 >>> pi==float(Pi)
6 True
```

3 Programmation

3.1 Introduction à la notion d'algorithme

Il y a bien sûr des algorithmes partout dans l'informatique. C'est la base du fonctionnement de tout système numérique aujourd'hui. Mais en considérant la notion d'algorithme dans sa version la plus générale, il n'y a en a pas que dans les systèmes informatiques. On utilise des algorithmes dans la vie quotidienne sans s'en apercevoir.

Définition 3.1 (Algorithme) :
Ensemble de règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations. Un algorithme peut être traduit, grâce à un langage de programmation, en un programme exécutable par un ordinateur. *(Larousse)*

Plus clairement (et plus scientifiquement), un algorithme est une suite d'instructions qui doit dépendre de paramètres (appelés entrées) et donner un résultat (appelé sortie). L'algorithme doit comporter un nombre FINIE d'étapes de calculs et chaque étape doit être définie avec précision (au sens informatique du terme).

Exemple 3.1 :
La notice de montage d'un meuble IKÉA est un exemple d'algorithme. Les entrées sont les pièces du kit et le résultat, est le meuble. Et chaque étape est clairement indiqué de façon très précise. Et il n'y a qu'un nombre fini de page à la notice.

Une recette de cuisine est un autre exemple classique d'algorithme.

!!! ATTENTION !!!

⚠ Un algorithme est une suite d'instructions finie. Ce qui sous-entend que pour montrer qu'une suite d'instructions est algorithme, il y a deux choses à faire : Montrer qu'on a ce qu'on veut à la fin ; montrer que c'est en un temps fini.

Remarque :
On appelle pseudo-code la description d'un algorithme dans un autre langage qu'un langage informatique. C'est une sorte de version de l'algorithme dans le langage courant.

Définition 3.2 (Programme) :
Ensemble d'instructions et de données représentant un algorithme et susceptible d'être exécuté par un ordinateur. *(Larousse)*

Un programme est donc la traduction d'un algorithme dans un langage compréhensible par l'homme et interprétable par la machine. C'est un ensemble d'instructions regroupés dans un fichier appelé le code source du programme.

Il y a deux types d'instructions :

- Les instructions simples. Ce sont les tâches qui se font directement. Comme la déclaration d'une variable, l'affectation d'une valeur à une variable. En Python, ces deux opérations se font en même temps. Par exemple,

```
1 x=2
```

- Les instructions composées. Ce sont les tâches plus complexes qui en imbriquent d'autres qui seront effectuées selon certains critères. Il y a 3 types d'instructions composées :

1) Les séquences. C'est un regroupement d'instructions les unes à la suites des autres qui vont être exécutées dans l'ordre. Par exemple

```
1 x = 2
2 x = x+1
3 x = x*2
```

2) Les instructions conditionnelles, appelés plus généralement boucle if/else que nous verrons plus en détails juste après. Le principe est de donner différentes instructions qui seront exécutées selon certaines conditions.
3) Les boucles. Ce sont des suites d'instructions qui sont répétées en boucle un certain nombre de fois. Il y a deux types de boucles : les boucles for et les boucles while. Nous allons voir ces boucles en détails aussi.

3.2 Les fonctions

Les fonctions et procédures sont les briques constitutives de l'outil informatique. Elles permettent de décomposer un programme complexe en plein de petits sous-programme plus simple pouvant être réutilisé à volonté.

3.2.1 Définition et Syntaxe

Définition 3.3 (Fonction (informatique)) :
Une fonction (informatique) est une suite d'instruction dépendant éventuellement de paramètres, regroupés sous un nom pouvant être exécuter à la demande. La syntaxe pour définir une fonction est de commencer par le mot-clé **def** suivi du nom de la fonction avec les paramètres. Le corps de la fonction doit être indenté. La fin de la fonction étant obtenu dès la fin de l'indentation.

```
1 def fonction(var1:type1 ,var2:type2 ,..., varn:typen) -> typef :
2     ligne1
3     ligne2
4     ligne3
5     ...
6     lignep
```

Les paramètres dont dépend la fonction sont appelés paramètres d'entrée. Les éléments que renvoie la fonction sont appelés paramètres de sortie.

Les paramètres sont optionnels. Il est possible de faire une fonction qui ne dépende d'aucunes variables. Dans ce cas, les parenthèses dans le nom de la fonction sont laissées vide (mais elles doivent apparaître).

L'annonce des types des différentes variables et de la sortie n'est pas obligatoire mais recommandée pour avoir les idées claires sur la nature de chaque variables.

Le nom d'une fonction répond presque aux mêmes critères que le nom d'une variable. C'est un mot arbitraire composée uniquement de lettres (minuscules ou majuscules) et de chiffres sans espaces et de tirets (du haut ou du bas). L'idéal étant de choisir ce mot de façon à ce qu'il décrive ladite fonction. Comme pour les variables.

Remarque :
Une fois la fonction créée dans l'éditeur et compilé, il ne fait pas oublier d'utiliser la fonction dans l'interpréteur. Python ne fait rien tout seul. Ce n'est pas parce que la fonction est créée qu'elle est utilisée. Ce n'est pas parce que vous avez demandé à Python de fabriquer un algorithme qu'il peut comprendre de lui même qu'il faut utiliser cet algorithme. C'est donc à vous de l'utiliser dans l'interpréteur (ou dans l'éditeur).

Exemple 3.2 :
La suite de Fibonacci est la suite définie par

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ \forall n \in \mathbb{N},\ u_{n+2} = u_n + u_{n+1} \end{cases}$$

Pour faire une fonction qui calcule le n-ème terme, on pourrait faire

```
1  def Fibonacci(n:int) -> int :
2      x1=0
3      x2=1
4      x=1
5      for i in range(2,n+1):
6          x=x1+x2
7          x1=x2
8          x2=x
9      return(x)
```

et pour calculer le 10-ème terme, il suffirait de faire

```
1  >>> Fibonacci(10)
2  55
```

Une fonction est donc une sorte de boîte noire. Elle s'apparente un peu aux fonctions mathématiques (mais ATTENTION ! Il ne faut pas confondre les deux !).

Définition 3.4 (`return()`) :
Pour renvoyer le résultat d'un calcul d'une fonction pour exploitation ultérieure, on utilise la commande `return(var)`. C'est la valeur de la variable `var` qui est renvoyée et pas le nom.

Vous avez déjà pu observer dans le TP0 que si l'on demande d'effectuer un calcul à partir de l'éditeur, juste un calcul, rien n'est affiché sur l'interpréteur. L'ordinateur a effectivement fait le calcul. Mais il ne faut pas confondre faire un calcul et afficher le résultat. Python ne prend de décision par lui même. Si vous ne luis demandez pas d'afficher le résultat d'un calcul, il ne l'affichera pas. D'où l'intérêt de la commande `return` qui permet de pouvoir renvoyer le résultat d'un algorithme.

Remarque :
Un programme ne peut renvoyer qu'une seule valeur. Il ne peut pas en renvoyer plus. On ne peut manipuler qu'une seule chose à la fois. Aussi, toutes les instructions qui se trouvent à l'intérieur d'un programme après un `return()` ne sont pas effectuées. Le `return` contient un `break()` qui arrête l'algorithme (voir chapitre 3 Algorithmie).

Si un algorithme calcule plusieurs choses en même temps, il faudra tout retourner d'un seul coup dans le même `return()` en séparant les différents éléments par des virgules.

!!! ATTENTION !!!

⚠ Il ne faut pas confondre les commandes `print` et `return`. La commande `print()` ne fait qu'afficher quelque chose. Elle n'arrête pas l'algorithme. Et on ne peut pas réutilisé ce qui est affiché (cf `type(print(3*2))`). Ce n'est qu'un bulletin d'information.

Exemple 3.3 :

```
1  def Somme(n:int) -> int:
2      S=0
3      for k in range(1,n) :
4          S=S+k
5          return(S)
6  def Somme2(n:int) -> int:
7      S=0
8      for k in range(1,n) :
9          S=S+k
10         print(S)
```

et on obtient

```
1  >>> Somme(5)
2  1
3  >>> s=Somme(10)
4  >>> s
5  1
6  >>> Somme2(5)
7  1
8  3
9  6
10 10
11 >>> s=Somme2(3)
12 1
13 3
14 >>> s
15 >>> print(s)
16 None
```

On ne peut donc pas exploiter le résultat de la fonction `Somme2` et la fonction `Somme` ne fait qu'un seul tour. Elle s'arrête dès la première exécution du `return`.

3.2.2 Variables locales vs Variables globales

3.2.2.1 Variables locales

> Définition 3.5 (Variable locale) :
> Une variable locale est une variable qui ne sert que localement, dont la définition n'est circonscrite qu'à un domaine fini du code, un certain nombre de lignes de code.

Typiquement, une variable locale est une variable qui apparaît dans un algorithme. Ce qui est dans une fonction reste dans la fonction. On ne définit dans une fonction que ce dont on a besoin pour la fonction. Toutes les éventuelles variables utilisées sont réinitialisées dès que l'on sort de la fonction. Il faut alors les redéfinir. On peut ainsi utiliser le même nom pour une variable dans différentes fonctions sans peur d'interférences.

Exemple 3.4 :
On peut écrire les deux fonctions suivantes avec les mêmes variables locales et il n'y a pas d'interférences.

```
1 def somme(n:int) -> float :
2     S=n*(n+1)/2
3     return(S)
4 def sommeCarre(n:int) -> float:
5     S=n*(n+1)*(2*n+1)/6
6     return(S)
```

et on peut les appeler par

```
1 >>> Somme(10)
2 55.0
3 >>> SommeCarre(10)
4 385.0
5 >>> S
6 NameError: name 'S' is not defined
```

Le variable S n'est pas défini. Elle n'existe pas. Et il n'y a pas de problème entre les deux fonctions bien qu'elles utilisent le même nom de variable locale.

3.2.2.2 Variable globale

Définition 3.6 (Variable globale) :
Une variable globale est une variable qui est valable en n'importe quel point du code. C'est une variable qui ne dépend pas de l'endroit du code où on se trouve. Elle est défini de façon globale sur le code.

Typiquement, une variable globale est une variable définie en dehors d'une fonction. Comme elle est définie en dehors de toutes fonctions, elle existe tant que le système n'est pas remis à zéro. Elle pourra donc être aussi utilisée dans une fonction mais avec sa valeur en tant que valeur globale. La modification de cette variable globale au sein d'une fonction pourrait entraîné des conflits pour la suite. La modification de sa valeur dans une fonction entraîne une modification de sa valeur globalement, sur tous le code. C'est la variable globale compète qui change. La modification n'est pas locale.
Exemple 3.5 :

```
1 P=5
2 def Produit(n:int) -> int :
3     P=P*n
4     return(P)
```

La compilation se passe bien, mais on obtient un message d'erreur lors de l'appel de la fonction produit avec n'importe quelle valeur pour n :

```
1 >>> Produit(2)
2 UnboundLocalError : local variable 'P' referenced before assignment
```

Ce message d'erreur veut dire que Python prend la variable `P` pour une variable locale et comme sa valeur n'est pas initialisée, Python ne peut pas lancer le calcul. Il confond variable locale et variable globale. Par définition, lors d'affectation, Python prend toutes les variables comme des variables locales.

Mais on peut quand même se servir d'une variable globale en la considérant comme un paramètre :

```
1 P=5
2 def Produit(n:int) -> int:
3     PP=P*n
4     return(PP)
```

et là tout va bien :

```
1 >>> Produit(5)
2 25
3 >>> Produit(10)
4 50
5 >>> P
6 5
```

Il faudra donc faire très attention aux noms des variables que l'on utilise. On ne gère pas de la même façon les variables locales et les variables globales. Les noms des variables globales devront être choisis avec grand soin pour ne pas être gêné par la suite ; le nom des variables globales doit être cohérent sur l'ensemble du code. En revanche, les noms des variables locales ne doivent être cohérent qu'à l'intérieur de la fonction où elles sont circonscrites. On a donc plus de flexibilité à ce niveau là (en prenant garde de ne pas avoir de problème avec les variables globales).

Définition 3.7 (`global`) :
Il est possible d'utiliser une variable globale dans une fonction et de modifier sa valeur. Pour se faire, il faut spécifier que la variable globale à l'aide de la commande

```
global var1, var2, ...., varn
```

Attention, si la valeur des variables globales sont modifiés dans la fonction, elles le resteront en dehors.

Attention, il n'y a pas de parenthèse avec la commande `global`. Mettre des parenthèses renverrai une erreur.

Exemple 3.6 :

```
P=5
def Produit(n:int) -> int :
    global P
    P=P*n
    return(P)
```

ce qui nous donne

```
>>> P
5
>>> Produit(5)
25
>>> P
25
```

3.2.3 Documentation

On rappelle qu'il est utile de commenter les lignes de codes afin de se souvenir de ce dont on souhaite faire pour pouvoir relire le code plus facilement ultérieurement.

Il peut être utile aussi de documenter une fonction.

Définition 3.8 (Documentation d'une fonction) :
La documentation d'une fonction est un texte (donc une chaîne de caractères) intégré à la fonction et qui est renvoyé lors de l'utilisation de la commande `help()` sur le nom de cette fonction.

La documentation est une chaîne de caractères entre triple guillemets insérés juste après le nom de la fonction.

Le but de la documentation est, un peu à l'instar des lignes de commentaires, d'afficher un court texte permettant de rappeler ce que fait la fonction et aussi de donner d'expliquer son utilisation. Et cette fois-ci, il n'y a pas le choix, la chaîne de caractères doit impérativement être comprise entre trois guillemets.

Exemple 3.7 :
On reprend les codes des sommes

```
def Somme(n:int) -> float :
    """Permet de calculer la somme des n premiers entiers.
    n -- entier"""
    S=n*(n+1)/2
    return(S)
```

Ce qui permet d'avoir une aide en faisant

```
>>> help(Somme)
Help on function Somme in module __main__:

Somme(n)
    Permet de calculer la somme des n premiers entiers.
    n -- entier
```

3.2.4 Procédures

Les procédures et les fonctions sont identiques dans leur définition mais ne diffère que dans leur rendus. Pour faire court, la fonction calcul quelque chose et rend une valeur, la procédure fait quelque chose mais ne rend rien. La procédure fait une modification, affiche quelque chose etc.

Exemple 3.8 :

```
#Ceci est une fonction
def fctttc(val:float, taux:float) -> float :
    ttc=val*(1+taux/100)
    return(ttc)
#Ceci est une procédure
def procttc(val:float, taux:float) -> None :
    ttc=val*(1+taux/100)
    print(ttc)
```

3.3 Interactivité

Définition 3.9 (`input()`) :
Il est possible de demander à un algorithme d'interagir avec l'utilisateur grâce à la commande `input`. La syntaxe est

```
input("Invitation d'interaction")
```

Lors de l'apparition de cette commande dans un algorithme, l'interpréteur affiche alors la chaîne `"Invitation d'interaction"` et l'ordinateur attend une réaction de l'utilisateur. Ce que tape l'utilisateur est alors convertit en chaîne de caractère. Il faut donc penser à sauvegarder cette chaîne dans une variable.

Exemple 3.9 :

```
>>> L=input("enter quelque chose\n")
entrer quelque chose
blablubli
>>> L
'blablubli'
```

Cette commande est très utile pour faire des algorithmes interactifs, des petits jeux.

Exemple 3.10 :
La procédure suivante permet de jouer au jeu des bâtons de Fort Boyard : enlever de 1 à trois bâtons à chaque tour ; le joueur qui enlève le dernier bâton à perdu (on peut faire des versions où le joueur perd tout le temps).

```
def FortBoyard() -> None:
    import random as rand
    L="|"*20
    print(L)
    NbCoupJoueur=[]
    NbCoupOrdi=[]
    Triche=False
    while len(L)>0 :
        print("Il reste ",len(L)," batons")
        print("A vous de jouer")
        Joueur=int(input("Enlever combien de batons ? (1, 2 ou 3)\n"))
        if Joueur not in [1,2,3] :
            print("Vous trichez !! Vous avez perdu.")
            Triche=True
            break
        NbCoupJoueur=NbCoupJoueur+[Joueur]
        L=L[0:len(L)-Joueur]
        if len(L)==0 :
            print("Vous avez perdu !!")
            break
        else :
            print("A l'ordinateur de jouer.")
            Ordi=rand.randint(1,min(len(L),3))
        NbCoupOrdi=NbCoupOrdi+[Ordi]
        L=L[0:len(L)-Ordi]
        print("L'ordinateur enleve ",Ordi," baton(s)")
        print(L)
    if len(NbCoupJoueur)<=len(NbCoupOrdi) and Triche==False:
        print("Vous avez gagne !! Bravo !")
```

!!! ATTENTION !!!

Capytale (que nous utilisons cette année) est très bien, mais gère mal la fonction `input`. Ce n'est pas idéal, mais il faudra forcer le site à utiliser la fonction `input`. Pour cela, deux mots clés devront être rajoutés. Ils ne font pas partie de votre code. C'est seulement pour contourner les limitations de codage du site utilisé.

Il faudra rajouter le mot clé **`async`** avant le `def` de la fonction utilisant le `input`. Et il faudra rajouter **`await`** juste avant l'utilisation de la fonction.

Par exemple :

```
async def bonjour() :
    prenom=input("Quel est votre prénom ?\n")
    print("Bonjour "+prenom+" !")
>>> await bonjour()
```

3.4 Instruction conditionnelle (`if/else`)

Définition 3.10 (Instruction conditionnelle `if/else`) :
L'instruction conditionnelle consiste à faire quelque chose si une condition est vérifiée, et autre chose sinon. La syntaxe est

```
if condition_1 :
    instruction_1
else :
    instruction_2
```

La **`condition_1`** devant être obligatoirement un booléen. Si ce booléen est `True`, alors c'est **`instruction_1`** qui est effectuée et dans le cas contraire, c'est **`instruction_2`** qui est effectuée.

Exemple 3.11 :

```
def fct(n:int) -> int : # Définir fonction fct
    if n%3==0 :         # Si n est divisible par 3, faire
        return(n//3)        # Renvoyer n/3
    else :              # Sinon, faire
        return(3*n)     # Renvoyer 3*n
```

Il est bien sûr possible de composer les structure conditionnelle :

Exemple 3.12 :

```
def fct(n:int) -> int :          # Définir la fonction fct
    if n%3==0 :                  # Si n est divisible par 3
        return(n/3)              # Renvoyer n/3
    else :                       # Sinon, faire
        if (n-1)%3==0 :          # Si n-1 est divisible par 3, faire
            return((n-1)//3)         # Renvoyer (n-1)/3
        else :                   # Sinon
            return((n-2)//3)         # Renvoyer (n-2)/3
```

!!! ATTENTION !!!

⚠ Attention à l'indentation. Un même bloc d'instructions ayant la même place dans la hiérarchie de dépendance doit avoir la même indentation.

Définition 3.11 (elif) :
Il est possible de mettre plusieurs conditions à la suite avec la structure conditionnelle. Le principe est le suivant : si quelque chose se passe, alors on fait on fait ça ; et sinon, s'il se passe ça, on fait ceci ; et sinon s'il se passe cela, on fait ça ; etc ; et sinon, on faut autre chose :

```
if condition_1 :
    instruction_1
elif condition_2 :
    instruction_2
elif condition 3 :
    instruction_3
...
else :
    instruction_n
```

Exemple 3.13 :

```
1  def fct(n:int) -> int :       # Définir la fonction fct
2      if n%3==0 :               # Si n est divisible par 3, faire
3          return(n//3)          # Renvoyer n/3
4      elif n%3==1:              # Si n-1 est divisible par 3, faire
5          return((n-1)//3)      # Renvoyer (n-1)/3
```

Remarque :
Il n'est pas nécessaire de mettre un `else`. Ne pas le mettre revient à dire que si on est pas dans les cas précédents, alors on ne fait rien.

Exemple 3.14 :
La suite de Syarcuse est définie par

$$\begin{cases} u_0 = a \in \mathbb{N} \\ \forall n \in \mathbb{N},\ u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ pair} \\ 3u_n + 1 & \text{sinon} \end{cases} \end{cases}$$

Écrire une fonction `Syra(u:int) -> int` dépendant d'un paramètre `u` et permettant de calculer un terme de la suite connaissant son prédécesseur `u`.

3.5 Boucles (`for` et `while`)

Définition 3.12 (`range(n,m,p)`) :
La commande `range` permet de pouvoir lister des entiers. Il y a trois façons d'utiliser `range` :

- `range(n,m,h)` avec trois paramètres. C'est la version la plus complète. C'est une liste contenant tous les termes allant de `n` à `m-1` avec un pas de `h`. Autrement dit, la première valeur est n, puis $n+h$, puis $n+2h$ etc jusqu'à $m-1$.
- `range(n,m)` avec 2 paramètres. C'est la version intermédiaire. On ne donne que le point de départ et d'arrivée et le pas est implicitement de 1. Il correspond donc à `range(n,m,1)`.
- `range(n)` avec 1 paramètre. C'est la version courte. Il correspond à `range(0,n,1)`. C'est donc la liste de tous les entiers compris entre 0 et $n-1$.

⚠ On rappel que python commence à compter à partir de 0. Donc le dernier index est toujours un cran avant le dernier indiqué. Une autre façon de le dire est que python commence à compter à 0 les choses sont faites pour que `range(0,n)` contiennent n termes. Il faut donc s'arrêter à $n-1$.

3.5.1 Boucle `for`

Définition 3.13 (Boucle `for`) :
Une boucle **`for`** est une boucle permettant de répéter des instructions un certains nombres de fois. Il y a un compteur intégré qui compte le nombre de tour.

```
for k in range(n,m) :
    instructions
```

A chaque fois que l'ordinateur a atteint la fin des instructions de la boucle for, il remonte au début, incrémente la valeur de **`k`** et recommence.

Exemple 3.15 :

```
def Carre(n:int, m:int) -> int : # Définir la fonction Carre
    S=0                          # Initialisation de la somme à 0
    for k in range(n,m+1) :      # Pour k allant de n à m
        S=S+k**2                 # Ajouter k^2 à S
    return(S)                    # Renvoyer S
```

Exemple 3.16 :
Écrire un algorithme `Facto(n:int) -> int` permettant de calculer $n!$.

Remarque :
Il y a des alternatives à la syntaxe de la boucle **`for`**. il est possible de remplacer le `range` par n'importe quel itérable, c'est à dire par n'importe quoi pouvant être parcouru. On peut donc mettre une liste, une chaîne de caractère ...

```
def ComptageLettre(chaine:str) -> None :
    C=chaine.lower()
    alpha="abcdefghijklmonpqrstuvwxyz"
    for lettre in alpha :
        compteur=0
        for l in C :
            if l==lettre :
                compteur = compteur+1
        print("il y a ",compteur,"lettre ",lettre)
```

3.5.2 Boucle while

Définition 3.14 (Boucle **while**) :
Une boucle while est une suite d'instructions devant être faites tant qu'une certaine condition est remplie.

```
while condition :
    instructions
```

!!! ATTENTION !!!

⚠ Il faudra TOUJOURS prendre garde que la boucle s'arrête à un moment donné. Si l'on met une condition qui est toujours vraie, la boucle est infinie. Ce qui est problématique. Il faudra donc toujours s'assurer que la condition fini par devenir fausse à un moment donné.

Dans la pratique, il faut que la **condition** change à chaque itération de la boucle while. Il faut qu'à chaque nouveau passage, sa valeur change pour qu'il y ait une chance que la **condition** devienne fausse. Mais ça ne suffit pas.

Remarque :
Dans le cas où l'on obtient une boucle infinie, l'ordinateur va calculer indéfiniment. Il faut donc l'arrêter. Pour ce faire, en haut à droite de l'interpréteur, il y a un bouton prévu à cet effet. Il faut sélectionner "Interrompre le noyau". Ça arrête les calculs en cours. Il faut ensuite relancer le noyau avec "Redémarrer le noyau". Il faudra alors recharger tous les modules. Le redémarrage du noyau consiste à tout effacer et tout recommencer. On redémarre à neuf. Les variables globales sont effacées, et tout ce qui était écrit dans l'interpréteur est perdu. Seul ce qui est dans l'éditeur est conservé. Mais qu'il faudra recompiler pour pouvoir utiliser les algorithme qui y sont sauvegardés.

Exemple 3.17 :

```
def puissance2(n:int) -> float :     # Définir la fonction fct dépendant du paramètre n
    p=0
    while n%2==0 :       # Tant que n est divisible par 2, faire
        n=n//2           # n est divise par 2
        p=p+1
    return(p)            # Renvoyer la valeur de p
```

Cet algorithme donne donc la plus grande puissance de 2 divisant n.

Exercice 1 :
Si $a \in \mathbb{R}_+^*$, la suite

$$\begin{cases} u_0 = 1 \\ \forall n \in \mathbb{N},\ u_{n+1} = \frac{1}{2}\left(u_n + \frac{a}{u_n}\right) \end{cases}$$

converge vers $\sqrt{a}$. Faire une fonction qui prenne en argument un réel a et un réel (petit) ε et qui renvoie le nombre de pas à faire pour que $|u_n - \sqrt{a}| \leq \varepsilon$.

3.5.3 Comparaison `for` vs. `while`

La différence entre une boucle **`for`** et une boucle **`while`** est essentiellement une différence d'intérêt. Dans le cas d'une boucle **`for`**, on s'intéresse au nombre de tour effectué dans la boucle, qu'on impose. On obtiendra alors le résultat qu'on obtiendra. On devra faire avec. La boucle **`while`** s'intéresse plutôt au résultat : elle impose de ne s'arrêter que lorsqu'un résultat attendu est atteint. Et on devra tourner dans la boucle autant qu'il le faut pour ça.

Il est alors (théoriquement, mathématiquement) possible de passer de l'un à l'autre. Si l'on connaît le résultat d'une boucle **`for`**, on peu la transformer en une boucle **`while`**. Et inversement, si on connaît le nombre de tour nécessaire pour sortir d'une boucle **`while`** (à l'aide d'un compteur, par exemple), on peut alors la transformer en une boucle **`for`**.

Néanmoins, en pratique, on ne peut pas toujours le faire (il n'est pas toujours possible de savoir à l'avance le nombre de tour nécessaire dans une boucle, par exempele).

Exemple 3.18 :
Les deux codes suivants permettent de calculer la somme de tous les cubes inférieur à n :

```
def SommeCubeWhile(n:int) -> int :
    S=0
    k=0
    while k**3<n :       # Tant que le cube est inférieur à n
        S=S+k**3         # Ajouter k^3 à S
        k=k+1            # Prendre l'entier suivant
    return(S)            # Renvoyer S
```

et la même version avec une boucle for :

```
def SommeCubeFor(n:int) -> int :
    S=0
    for k in range(0,int(n**(1/3.0))+1) :    # Pour k allant de 1 à ⌊√n⌋ faire
        S=S+k**3                             # Ajouter k^3 à S
    return(S)
```

3.6 Fonctions mathématiques

Les objets mathématiques de bases ($\exp$, $\ln$, $\cos$, $\sin$, π, etc) sont déjà implémentés dans Python dans le package **math**. Pour pouvoir utiliser ces fonctions, il faudra donc au préalable importer le module **math** et utiliser les fonctions mathématiques qui se trouvent à l'intérieur.

L'importation d'un module se fait par la commande **import**. Pour utiliser une fonction (ou que ce soit) dans un module, il faut utiliser la commande **nomModule.nomFct()**. Pour simplifier les notations, lors de l'importation d'un module, on peut lui donner une alias avec la commande **as**. L'alias est une étiquette permettant de renommer un module.

Exemple 3.19 :

```
>>> cos(pi)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'cos' is not defined
>>> 2*pi
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'pi' is not defined
>>> import math as mt
>>> mt.cos(mt.pi)
-1.0
>>> 2*mt.pi
6.283185307179586
>>> mt.e
2.718281828459045
>>> math.sqrt(4)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'math' is not defined
>>> mt.sqrt(4)
2.0
```

Par conséquent, on oubliera pas d'importer le module **math** en début de script si besoin. On

prendra garde aussi à n'importer un module qu'une seule fois pour ne pas surcharger la mémoire de l'ordinateur.

4 Débogage

Il est fréquent de tomber sur des erreurs lors de l'élaboration d'un programme. C'est l'interpréteur qui renvoie les messages d'erreurs en rouge. Il est utile de savoir les interpréter pour pouvoir avoir des indications (partielles) sur l'erreur. Même avec ces indications, il n'est pas toujours aisé de corriger l'erreur.

Les messages d'erreur les plus fréquents sont les suivants :

- `SyntaxError: invalid syntax`
  Problème de syntaxe. L'interpréteur ne reconnaît pas la structure syntaxique de l'algorithme. Souvent à cause d'une faute de frappe. Un mot clé est mal orthographié, une parenthèse manquante, un deux point oublié ...
- `NameError: name 'var'is not defined`
  L'interpréteur ne trouve aucune référence au mot `var`. Il ne sait pas à quoi il correspond. Il s'agit en général d'une variable, d'un module ou d'une fonction n'ayant pas encore été défini.
- `TypeError: can only concatenate list (not "int") to list`
  Ce message a beaucoup de variante. Cette erreur apparaît à chaque fois que l'on essaie de faire des opérations entre deux objets de types différents ne pouvant pas être additionné. Par exemple, l'addition d'une liste et d'un entier, d'une liste et d'une chaîne etc.
- `TypeError: 'list' object is not callable`
  Là encore, plusieurs variante pour cette erreur. Elle survient lorsque l'interpréteur tente de manipuler un objet comme une fonction et qui n'en est pas. En général, cette erreur survient en tentant d'accéder à un terme d'une liste en mettant des parenthèses au lieux des crochets.
- `IndexError: list index out of range`
  Cette erreur survient lorsque l'on dépasse la capacité d'une liste. Typiquement, lorsque l'on demande à l'interpréteur d'accéder à un numéro d'élément d'une liste qui dépasse le nombre d'élément de la liste.
- `ZeroDivisionError: division by zero`
  No comment.
- `IndentationError: unexpected indent`
  Problème dans les indentations du code. Souvent du à cause d'un espace (touche espace) qui s'est intercalé au début d'une ligne faisant un petit décalage.

Exemple 4.1 :
Faire un algorithme `Avocat(chaine:str) -> str` qui permet de coder une chaîne de caractère avec le principe "A vaux K", *i.e.* A⇝K, B⇝L, C⇝M, etc.

Définition 4.1 (La commande `assert`) :
La commande **`assert condition`** permet de lever une erreur dans un programme. C'est une ligne utilisé dans le cadre de débogage qui permet d'opérer un contrôle sur les variables. Si la condition de l'assertion est vérifiée, rien ne se passe et le programme continue. Si la condition est fausse, le programme s'arrête et une message d'erreur est renvoyé.

Exemple 4.2 :

```
1 def discriminant(a:float, b:float, c:float) -> float :
2     assert a != 0    # si a=0, le code s'arrête et renvoie AssertionError
3     return(b**2-4*a*c)
4 >>> discriminant(1,1,1)
5 -3
6 >>> discriminant(0,1,1)
7 AssertionError
```