

# Python

*Dans ce document nous allons retracer toutes les notions essentielles en Python. Tout ceci doit bien être maîtrisée pour assurer une bonne maîtrise du langage. Nous allons voir de manière détaillée toutes les choses les plus insignifiantes pour bien s'imprégner du langage de programmation, alors si cela te paraît ennuyant ou nul, alors tant mieux ! Cela veut dire que tout est vraiment clair dans ta tête et ça c'est mega cool. Ce cours se base sur mes connaissances personnelles, rien de plus.*

## BASES

*Comment parler de langage de programmation sans parler d'expression. Qu'est-ce qu'une expression en langage de programmation, et plus précisément en Python ? Une expression n'est rien d'autre qu'une suite de caractères définissant une valeur. Afin de déterminer cette valeur, ton ordinateur doit évaluer cette expression. Afin de mieux, comprendre voici un exemple d'expression en Python.*







```
10 + 11
#Cela affiche 21

5//2 + 3
#Cela affiche 5

3.5 + 2.5
#Cela affiche 6.0
```

*Si tu remarques bien, chaque résultat a une forme différente. Le premier donne 21, et le troisième donne 6.0. Pourquoi 21 n'est pas écrit sous la forme 21.0 ? Eh bien ici se trouve la notion des types de variables !*

*Il existe plusieurs types différents, voici les principaux :*

-  Entiers (int)
-  Flottant (float)
-  Booléen
-  Chaînes de caractères (string)
-  Listes
-  Fonctions

*J'ai mis des petits ordinateurs pour faire geek !*

*Bien définir les types de variables est très important ! En effet, cela va affecter le résultat qui pourra être retourné et les opérations que nous pourrions réaliser. Cela a également un impact sur la mémoire prise par la variable mais ici, ce n'est pas le sujet (toujours bon à savoir).*

*Je ne sais pas si tu le savais, mais il est tout à fait possible de demander à python le type d'une expression !*

```
type(21)
# <class 'int'>

type(6.0)
# <class 'float'>

type(True)
# <class 'bool'>

type("Vous êtes beau")
# <class 'str'>
```

```

type([1, 2, 3])
# <class 'list'>

type((1, 2, 3))
# <class 'tuple'>

type({1, 2, 3})
# <class 'set'>

```

Voici ce que Python te renverra en entrant cela dans le terminal ! Cela peut être très pratique pour vérifier le type d'un de tes résultats ou d'une de tes variables.

En effet, en Python, une expression a différentes formes. Elle peut être :

- ▣ Le résultat d'une fonction appliquée à une ou plusieurs expressions
- ▣ Le nom d'une variable
- ▣ Une constante
- ▣ La composée de plusieurs fonctions

Nous aurons l'occasion de revoir cela plus en détail. Maintenant que tu as ces bases, passons en revue les divers types que nous avons évoqués.

## Les entiers

Bon, ici rien de bien nouveau. Tout comme en mathématiques, nous retrouvons les entiers en Python. Ils sont de type « int ». Tu peux réaliser une série d'opérations avec ces entiers-là. En général, les entiers ne sont pas bornés sur Python. Tu peux représenter des nombres tout simplement gigantesques, à condition que ta machine suive... Les entiers sont les entiers relatifs  $\mathbb{Z}$ . Ainsi, tu peux représenter les entiers positifs et négatifs.

```

10 ** 39
# Affiche 1 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000

3 * -5
# Affiche -15

```

Voici un tableau de toutes les opérations que tu peux réaliser avec ce type :

opérations	+	-	*	//	%	**
signification	addition	soustraction	multiplication	Division entière	modulo	exponentiation

Attention, il peut être important de revenir sur l'opérateur de la division entière et celui du modulo.

Comme son nom l'indique, la division entière renvoie un entier. Lors de ton opération, tu vas diviser tes deux nombres entre eux et renvoyer l'entier inférieur.

Le module est très utile en arithmétique, il va te renvoyer le reste de ta division euclidienne.

```

10 % 3
# Affiche 1

```

```
10 // 3
# Affiche 3
```

Le reste de la division euclidienne de 10 par 3 est bien 1.

La division entière de 10 par 3 est bien 3. En effet, cette opération vaut 3.33333.... Ainsi, l'entier inférieur est bien 3.

Bien entendu, comme en mathématiques, nous ne pouvons pas diviser par 0 ! Ainsi,  $a \% 0$ ,  $a/0$  ou bien  $a//0$  ne sont pas définies. Ces expressions te renverront une erreur.

Attention ! Comme en mathématiques, il y a des priorités lors des opérations. Voici les priorités par ordre :

1. Exponentiation
2. Modulo
3. Multiplication et division entière
4. Addition et soustraction

Ainsi, dans une expression, nous allons d'abord calculer l'exponentiation, puis le modulo, puis les multiplications/divisions, puis nous finirons par les additions/soustraction. Pour mettre la priorité ailleurs, il faut alors mettre des parenthèses. Dans le cas de plusieurs parenthèses, nous commençons par celles qui sont imbriquées dans une autre parenthèse, sinon, nous y allons de gauche à droite.

```
5 + 6**2//4 - 6
# Affiche 8

(5+6)**2 // 4 - 6
# Affiche 24

(2+3) - (5*(4-2))
# Affiche -5
```

Bon, rien de bien passionnant, mais nous sommes obligés d'y passer !

## Les flottants

Il y a des choses bien plus intéressantes à dire sur la construction de ces nombres, en revanche, ici ce n'est pas l'objet du cours... Pour ta culture, nous codons sur 64 bits les flottants. Un bit sert pour le signe, 11 bits d'exposants et 52 bits de mantisse. En revanche, ici nous n'aborderons pas ces notions, c'est seulement pour la culture !

En tout cas, ces nombres ne sortent pas de nulle part. De cette construction, on peut majorer les nombres flottants. Le plus grand nombre positif est légèrement inférieur à  $2^{1024}$ . C'est grand, très grand mdr. Pour une simple comparaison, il y a environ  $10^{80}$  atomes dans l'Univers.  $2^{1024}$  c'est environ  $1.8 \times 10^{80}$ . Est-ce bon pour toi ? Voici les opérations possibles avec les flottants :

opérations	+	-	*	/	%	**
signification	addition	soustraction	multiplication	Division	modulo	exponentiation

Attention, la division entière n'est pas à utiliser avec les flottants, cela peut générer des erreurs. On utilise rarement le modulo, mais il est tout à fait possible de s'en servir.

Comme pour les entiers, il y a des priorités dans les calculs. Les voici :

1. Exponentiation
2. Multiplication et division
3. Addition et soustraction

```
2.5 + 2.5**2/2  
# Affiche 5.625
```

```
(2.5 + 2.5)**2 / 2  
# Affiche 12.5
```

**Important** : Lors d'opérations entre des entiers et des flottants, les entiers sont automatiquement convertis en flottants. Ainsi, le résultat retourné sera un flottant.

**Astuce** : Comment reconnaître si un résultat est un entier ou un flottant ? Il est vrai d'affirmer que 12.0 est égal à 12. En revanche, on voit que l'un a une décimale et l'autre non. Les entiers n'ont jamais de décimale. Ainsi, 12.0 est un flottant en Python et pas un entier.

## Les booléens

Ce type de variable est l'un des plus intéressants et des plus importants en informatique, tout langage confondu. Il s'agit d'un type binaire. Il n'y a que deux valeurs possibles :

1. True (Vrai)
2. False (Faux)

Cela paraît peu, mais c'est déjà bien assez. Toute la suite (les structures de contrôle) sera basée sur ce type. Cela nous permet de faire des vérifications.

Il existe également des opérations pour les booléens. Cela renvoie aux tables de vérités en mathématiques. Nous utilisons le même principe sur les prédicats. Ces opérations sont au nombre de trois : not, and, or. Voici la table de vérité :

<i>a</i>	<i>b</i>	<i>not a</i>	<i>a or b</i>	<i>a and b</i>
<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>
<i>False</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>False</i>
<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>

Voici un parallélisme avec les mathématiques.

On appelle assertion tout énoncé pouvant être VRAI(V) ou FAUX(F).

Soit *P* une assertion. On appelle négation de *P* l'assertion ayant les valeurs de vérités opposées. On la note  $\neg P$  ou **non**(*P*). On définit via la table de vérité :

<i>P</i>	$\neg P$
<i>V</i>	<i>F</i>
<i>F</i>	<i>V</i>

Soit  $P$  et  $Q$  deux prédicats. On appelle disjonction de  $P$  et  $Q$ , notée  $P$  ou  $Q$  (le `or` en Python) ou  $P \vee Q$  l'assertion étant vraie si et seulement si  $P$  est VRAIE ou  $Q$  est VRAIE. On dresse la table de vérité :

$P$	$Q$	$P \vee Q$
$V$	$V$	$V$
$V$	$F$	$V$
$F$	$V$	$V$
$F$	$F$	$F$

Soit  $P$  et  $Q$  deux assertions. On appelle conjonction de  $P$  et  $Q$ , notée  $P$  et  $Q$  (le `and` en Python) ou  $P \wedge Q$  l'assertion étant VRAIE si et seulement si  $P$  et  $Q$  sont toutes deux vraies. On dresse la table de vérité :

$P$	$Q$	$P \wedge Q$
$V$	$V$	$V$
$V$	$F$	$F$
$F$	$V$	$F$
$F$	$F$	$F$

Désolé pour cet écart, mais cela permet de voir le lien entre les deux, c'est toujours intéressant ! L'informatique est une application des mathématiques. Les opérations logiques sont le fondement d'énormément de chose, il est primordial de bien les maîtriser. Ces tableaux sont logiques, au sens mathématique du terme. Je t'invite à les relire autant que nécessaire pour t'y familiariser.

Comme pour les entiers ou les flottants, les booléens ont également des règles des priorités de calcul ! L'ordre est le suivant :

1. Not
2. And
3. Or

Nous pouvons utiliser des parenthèses si nécessaire.

```
False and False
# Affiche False

False and True or False
# Affiche False

False and False or True
# Affiche True

not True or not False and True
# Affiche True
```

Je t'invite à vérifier cela avec la table proposée au-dessus !

*Je pense ne rien t'apprendre en disant que l'on utilise que très peu les booléens sous cette forme. En réalité, nous allons plutôt « créer » les booléens ! Nous pouvons le faire via divers opérateurs de comparaisons et des opérateurs booléens !*

opérations	==	<	>	<=	>=	!=
signification	égal	inférieur	supérieur	Supérieur ou égal	Inférieur ou égal	Différent (non égal)

*Ces opérations sont très importantes car elles renvoient toutes des valeurs booléennes.*

```
3+4 == 7.0
# Affiche True

1 < 0
#Affiche False

0 != 10%2
# Affiche False car 10%2 vaut 0

10 >= 8
# Affiche True
```

*Lors d'opérations entre flottants et entiers, les entiers sont transformés en flottants. Cela est fait automatiquement par Python. L'exemple 1 démontre que les opérations de comparaisons n'ont pas la priorité sur les opérations arithmétiques. Nous effectuons d'abord les opérations arithmétiques puis les opérations booléennes.*

**Astuce :** si  $x$  est une variable booléenne, on écrira `not x` plutôt que `x == False`.

## Variables

*En Python, une variable est un nom qui est utilisé pour faire référence à une valeur stockée en mémoire. Les variables sont fondamentales dans la programmation car elles permettent de stocker et de manipuler des données.*

*Pour être plus formel, une variable est un identificateur auquel on associe une valeur. Un identificateur n'est rien d'autre qu'un nom, exceptées les suites de caractères dites «réservées». Elles sont réservées par Python. Par exemple, tu ne pourras jamais appeler une variable «if» ou bien « True » ou encore « return ». En revanche, tu peux l'appeler «je\_suis\_Adam» si tu le désires. Attention, les majuscules comptent. Ainsi, « adam » et « Adam » sont deux identificateurs différents. De plus, tu ne peux pas mettre d'espace. Par exemple, «je suis Adam» n'est pas un identificateur accepté par Python.*

*Une variable est donc un identificateur (un « nom ») auquel on assigne une valeur. Pour assigner une valeur, on utilise l'expression « = ». Il ne faut pas confondre avec == qui n'est pas une assignation mais une vérification (un test d'égalité).*

*Comment fonctionne une affectation ? L'expression à droite du égal est évaluée puis sa valeur est affectée à l'identificateur toujours présent à gauche du égal !*

```
x = 1 + 2
# x vaut 3. print(x) affiche 3

1 + 2 = x
# Ignominie...
```

*En Python, tu n'es pas obligé de déclarer ta variable en amont. En C++, il est nécessaire de d'abord initialiser les variables et leur assigner un type précis. Pour un entier, il faut avant déclarer « int x ». En Python, il n'y a rien à faire, c'est super ! Python déclarera la variable lors de sa première affectation et déduira le type tout seul.*

*Il est également important de noter que Python n'assigne pas un type précis à une variable. Une variable peut se voir assigner différent type au cours du programme*

```
x = 1+2
# x est de type int et vaut 3

x = x/2
# x est à présent de type float et vaut 1.5
```

*D'abord, x vaut 3. Ainsi, son type est ici un int par somme d'entier. Ensuite, on assigne à x sa précédente valeur divisée par 2. Ainsi, 3 divisé par deux vaut 1.5. Il s'agit donc d'un flottant. Il est important de noter l'indépendance entre plusieurs variables.*

```
x = 10
y = x
x = 5

print(y)
```

*Le résultat affiché est 10. Nous avons attribué à y la valeur de x, mais ceci se fait spontanément dans le temps. Cela ne veut pas dire que par la suite, y est nécessairement lié à x, non, loin de là ! Cela est très important car c'est une erreur récurrente lors des débuts.*

**Remarque :** *il ne peut pas y avoir d'opérateur dans le membre de gauche. Ainsi,  $x + 1 = 10$  ne veut rien dire en Python. Le résultat n'est pas  $x = 9$ . Le membre de gauche ne contient que des identificateurs, à droite tu peux raconter ta vie il n'y a aucun souci.*

*Un dernier concept est l'incréméntation (respectivement décréméntation). L'incréméntation fait référence à l'augmentation de la valeur d'une variable. Nous le réalisons grâce à « += ».*

*La décréméntation se fait quant à elle via « -= ». Elle permet de réduire la valeur d'une variable.*

```
x = 10
x += 2
# x vaut maintenant 12

x -= 4
# x vaut maintenant 8
```



## Structures de contrôle

*Les structures de contrôle en Python sont des mécanismes essentiels qui permettent de diriger le flux d'exécution d'un programme. Elles déterminent l'ordre dans lequel les instructions sont exécutées en fonction de conditions ou de répétitions.*

*Les deux principales structures de contrôle sont les structures conditionnelles et les boucles. Les structures conditionnelles, telles que « if », « elif » et « else », permettent d'exécuter des blocs de code en fonction de l'évaluation de conditions logiques. Cela permet de créer des chemins d'exécution différents en fonction des circonstances.*

*Les boucles, comme « for » et « while », permettent de répéter l'exécution d'un bloc de code jusqu'à ce qu'une condition spécifiée soit satisfaite.*

*L'indentation est cruciale dans Python pour délimiter ces blocs de code. L'indentation c'est le fait de laisser une marge devant un bloc d'instruction. L'indentation standard est d'une tabulation, soit 4 espaces.*

*Les structures de contrôle confèrent à un programme la flexibilité nécessaire pour s'adapter à différentes situations, améliorant ainsi la lisibilité et la logique du code. Elles sont fondamentales pour le développement de logiciels fonctionnels, permettant aux programmeurs de créer des applications plus complexes et puissantes.*

## Les instructions conditionnelles if/else

*Comme leur nom l'indique, ces instructions sont soumises à des conditions. Voici la structure générale pour les instructions conditionnelles :*

```
if condition1 :  
    [instructions à effectuer si la condition1 est vraie (True)]  
elif condition2 :  
    [instructions à effectuer si la condition2 est vraie (True)]  
else :  
    [instructions à effectuer si la condition1 et la condition2 sont fausses (False)]
```

*Les conditions utilisées sont de quel type ? Elles sont booléennes ! Ce n'est pas pour rien que je t'ai dit que ce type était probablement le plus important de tous... Ainsi, l'évaluation des conditions doit être un booléen. Pour ce faire, nous utiliserons des opérations de comparaisons et booléennes comme vues plus haut.*

*Dans ce type de structure conditionnelle, les conditions sont évaluées les unes après les autres, de haut en bas. Cela veut dire que nous allons d'abord vérifier la première condition, puis la seconde, puis la troisième, et ainsi de suite. Tu peux mettre autant de condition que tu veux, donc tu n'es pas limitée à deux. Ici, ce n'est que pour l'exemple. Finalement, le bloc else n'est exécuté que si toutes les expressions conditionnelles précédentes sont évaluées en false, donc si ces conditions sont fausses.*

```
if discriminant < 0 :  
    print("Il n'y a pas de solution réelle.")  
elif discriminant == 0 :  
    print("Il y a une solution réelle.")  
else :  
    print("Il y a deux solutions réelles.")
```

Voici un exemple qui évalue le nombre de solution(s) réelle(s) à une équation du second degré. Nous effectuerons un exercice plus tard qui reprendra toutes ces notions.

## Boucle conditionnelle while

La boucle *while* est une structure de contrôle en Python qui permet d'exécuter un bloc de code tant qu'une condition spécifiée n'est pas vraie. Elle est souvent utilisée lorsque le nombre d'itérations n'est pas connu à l'avance et dépend d'une condition.

On peut également le voir dans l'autre sens. La boucle *while* permet de réaliser une suite d'instruction tant qu'une certaine condition est vraie.

La structure est la suivante :

```
while condition :  
    [instructions]
```

Comment fonctionne cette structure ? Python va évaluer l'expression. Tant que la condition est vraie, donc que le résultat est *True*, on effectue toutes les instructions du bloc indenté (il est important de mettre l'indentation, sinon ça ne veut plus rien dire). Ensuite, une fois celles-ci achevées, on réévalue l'expression. Nous faisons ce processus tant que la condition est vraie, d'où le nom « *while* ».

```
compteur = 0  
while compteur < 5 :  
    print(compteur)  
    compteur += 1  
  
print("Boucle terminée")
```

Ce programme va afficher les entiers de 0 à 4. Comment fonctionne-t-il ? Le compteur est initialisé à 0. La boucle conditionnelle s'effectue tant que le compteur est strictement inférieur à 5. Si tel est le cas, nous affichons la valeur courante du compteur, puis nous y ajoutons 1. Nous reproduisons cela tant que le compteur est strictement inférieur à 5, donc jusqu'au cas où le compteur vaut 4. Lorsqu'il vaut 4, la condition est vérifiée. On affiche 4, et on incrémente de 1 la valeur du compteur. Voilà maintenant qu'il vaut 5 ! Cette fois-ci, la condition n'est plus vérifiée car 5 n'est pas strictement inférieur à 5, alors la boucle s'arrête car l'expression s'évalue en *False*.

Attention, rien n'oblige la boucle à s'effectuer ! Si la condition est initialement fausse, les instructions de la boucle *while* ne s'effectueront jamais !

```
compteur = 0  
while compteur > 1 :  
    print(compteur)
```

```
compteur += 1

print("Boucle terminée")
```

On initialise le compteur à 0 et la condition demande à ce que le compteur soit strictement supérieur à 1 pour effectuer les instructions. Cela n'est pas vérifié, et ce, dès le départ, alors on ne peut rien effectuer.

Le plus gros danger des boucles while est de mettre une condition qui est toujours vraie. Si tel est le cas, la boucle s'effectuera à l'infini, il faut donc être très vigilant sur cela, c'est l'erreur numéro 1 avec les boucles.

Nous avons vu que `False or True` vaut `True`, ainsi, en combinant ces deux conditions, dont l'une qui est toujours vraie, alors la boucle sera infinie par exemple !

```
x = 0
while x > 1 or x < 1 :
    x += 1
    print("Coucou")
```

Voici l'exemple d'une boucle infinie. Pour tout  $x$ , la disjonction est toujours vraie. En effet, nécessairement, nous aurons soit  $x > 1$  vrai et  $x < 1$  faux, ainsi, nous avons `True or False`, ce qui vaut `True` ; ou bien nous aurons  $x > 1$  faux et  $x < 1$  vrai, ainsi nous avons `False or True`, ce qui vaut aussi `True`. Quoi qu'il en soit, la condition est toujours vérifiée, ainsi la boucle sera infinie. C'est souvent plus subtil, mais il faut toujours penser à vérifier ces conditions !

## Boucle inconditionnelle for

La boucle conditionnelle while est très pratique lorsque l'on ne sait pas réellement combien de fois nous aurons besoin de répéter cette boucle. Par pallier ce souci, nous mettons alors une condition à vérifier. En revanche, parfois (dans la quasi-totalité des cas), on sait combien de fois nous souhaitons opérer une boucle. Par exemple, nous souhaitons afficher tous les entiers de 0 à 1000. Nous pouvons le faire avec une boucle while sans souci :

```
i = 0
while i <= 1000 :
    print(i)
    i += 1
```

Ceci est tout à fait valide, en revanche, nous préférons éviter ce type d'écriture. Puisque nous souhaitons à tout prix minimiser les erreurs, alors nous choisirons toujours la solution la plus simple et sûre.

Pour éviter ces complications (pour des codes bien plus complexes que celui-ci .. .), nous utiliserons des boucles inconditionnelles ! Comme son nom l'indique, ces boucles sont effectives sans conditions. Nous n'aurons plus d'intérêt à vérifier une condition pour effectuer la boucle.

```
for element in suite de valeur :
    [instructions]
```

La suite de valeur est appelée itérable. La variable « element » doit prendre successivement toutes les valeurs de la suite de valeur définie. Pour chacune de ces valeurs, nous exécuterons les instructions du corps de la boucle.

Un des itérables les plus communs est le constructeur « range ».

- Pour  $n \geq 0$ , `range(n)` fournit tous les entiers de 0 inclus à  $n$  EXCLU (on s'arrête donc à  $n-1$ , c'est très important de le notifier)
- Il est possible de commencer au-delà de 0. Par exemple, `range(m, n)`, avec  $m \leq n$  fournit les entiers de  $m$  à  $n-1$

Il est possible de modifier le pas d'itération.

- Si le pas est positif,  $p > 0$ , `range(m, n, p)` fournit les entiers allant de  $m$  à  $n-1$  d'un pas  $p$ .
- Si le pas est négatif, on inverse  $m$  et  $n$ , donc `range(n, m, p)`, avec  $n \geq m$ , fournit les entiers de façon décroissante de  $n$  à  $m$  via un pas de  $p$ .

Par exemple, si on veut traverser tous les nombres pairs entre 0 et  $m$ , nous pouvons utiliser `range(0, m, 2)`.

```
for i in range(0, m, 2):  
    print(i)
```

Cette boucle te renverra tous les entiers pairs compris entre 0 et  $m-1$ . Attention, tu dois définir une valeur de  $m$  avant de lancer cette boucle. De plus, un pas de deux ne veut pas dire que nous sélectionnons les entiers pairs ! Cela est vrai car nous allons de deux en deux ET que nous commençons par l'entier 0 ! Sans cela, si nous commençons à 1 ou 13 ou 71 ou 1001, n'importe quel nombre impair, alors nous parcourrions les entiers impairs.

Prenons également un exemple avec un pas négatif.

```
for i in range(10, -1, -1):  
    print(i)
```

Ici, nous allons afficher tous les entiers de 10 à 0 ! Pourquoi va-t-on de 10 à 0 et pas de 10 à -2 ? En effet, depuis tout à l'heure on dit que `range(n)` est exclusif, donc on va jusqu'à  $n-1$ . En réalité, pour ne pas te tromper, voyons la fonction `range` sous cette forme : `range(départ, arrivée, pas)`. Le premier argument est la position de départ dans notre boucle. On commence avec la valeur de départ. Si le pas est positif, nécessairement, on aura une suite de nombre croissante ! Ainsi, ta valeur d'arrivée doit être supérieure ou égale à la valeur de départ. En revanche, on va s'arrêter juste avant la valeur d'arrivée. D'où le fait qu'avec une suite de terme croissant, on va de la valeur de départ à la valeur d'arrivée - 1.

Lorsque le pas est négatif, on a une suite de nombre qui est décroissante. Ainsi, nécessairement ta valeur d'arrivée est inférieure à la valeur de départ. Cependant, comme pour le cas précédent, on s'arrête avant la valeur d'arrivée ! Or, comme la suite est décroissante, la valeur juste avant la valeur d'arrivée, donc à la valeur d'arrivée + 1 !

Donc, de façon générale, si le pas est positif (cas par défaut si tu ne précises pas l'inverse), alors tu vas jusqu'à  $n-1$ , sinon, si le pas est négatif, alors tu vas jusqu'à  $n+l$ , à condition de spécifier une valeur de départ supérieure à  $n$ .

La dernière boucle est très utile car elle permet de calculer les factorielles assez rapidement ! Voilà, c'était juste pour la culture mdr.

*L'itérable de ta boucle peut également être une liste ! Tu parcourras tous les éléments de ta liste, dans l'ordre établi par ta liste. Ainsi, ton petit « i », prendra à chaque étape la valeur de l'élément de ta liste.*

```
for element in [1,2,3,4,5,6,7,8,9,10]:  
    print(element)
```

*Cette boucle t'affichera tous les éléments de ta liste. Ainsi, tu verras s'afficher : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.*

*Cet aspect est vraiment pratique et un des plus utilisés. Nous verrons par la suite une autre méthode pour obtenir le même résultat.*

*Tu peux également parcourir les caractères d'une chaîne de caractère ! Tu parcourras toutes les lettres de ton mot si tu choisis un mot en particulier.*

```
for caractere in "J'aime les maths":  
    print(caractere)
```

*Ici, les espaces, les ponctuations, les symboles, tout est compté comme un caractère. Nous reviendrons plus en détail sur la chaîne de caractère par la suite. En tout cas, ici, tu verras s'afficher : J, ' , a, i, m, e, , l, e, s, , m, a, t, h, s. Tu verras que cela sera également très utile et on est souvent amené à les utiliser !*

*Je ne t'ai pas évoqué le cas du « break » et de « continue » dans les boucles. Si tu peux t'en passer, alors il faut t'en passer ! On n'aime pas trop utiliser ces instructions car tu as un plus grand risque d'erreur. En revanche, elles peuvent être très pratiques, notamment la fonction break. Dans une boucle while ou for, break va te permettre de sortir de la boucle lorsque tu le décides. C'est souvent relié à une condition à vérifier. L'instruction « continue », quant à elle, permet de passer à l'itération suivante en ignorant le reste du code dans le bloc.*

```
liste = [5, -3, 8, -1, 0, 2]  
  
for element in liste :  
    if element < 0 :  
        continue  
    print(element)
```

*Cette boucle parcourt tous les éléments de ta liste. Si ces éléments sont négatifs, alors on ne souhaite pas les afficher, sinon, on peut les afficher. On voit bien que si l'élément est négatif, alors on utilise continue, c'est-à-dire qu'on passe à l'itération suivante sans exécuter le reste du code, donc sans afficher l'élément ! Tu peux la tester dans ton Visual Studio Code. Elle t'affichera : 5, 8, 0, 2.*

*Voyons ce qu'il se passe si nous remplaçons l'instruction « continue » par l'instruction « break ».*

```
liste = [5, -3, 8, -1, 0, 2]  
for element in liste :  
    if element < 0 :  
        break  
    print(element)
```

Nous sortons du programme dès lors que nous rencontrons un élément négatif au sein de la liste. Typiquement, ici, la boucle s'arrêtera après avoir affiché 5. Une fois qu'elle va rencontrer le deuxième élément, ayant pour valeur -3, alors elle s'arrêtera car -3 est bien négatif.

## Boucles imbriquées

Le principe de boucles imbriquées est extrêmement important en informatique. Comme son nom l'indique, c'est une structure où tu as une boucle à l'intérieur d'une boucle. Très souvent, la boucle interne aura une itération qui dépendra de la valeur de la boucle extérieure. On peut faire une analogie en mathématique. Les boucles imbriquées ressemblent fortement aux sommes doubles. Essayons de calculer :

$$\sum_{i=1}^2 \sum_{j=1}^2 a_{ij}$$

Comment calculer cette somme ? Le principe n'est pas très difficile. On commence par regarder la somme externe, donc celle la plus à gauche. C'est une somme pour les éléments  $i$  allant de 1 à 2. Donc  $i$  ne prend que deux valeurs, 1 et 2. Ce que l'on va faire, c'est que nous allons fixer  $i=1$ . Maintenant que cela est fait, on voit ce que vaut la somme intérieure  $\sum_{j=1}^2 a_{ij}$ . On sait que nous sommes dans le cas où  $i$  vaut 1, alors, nous avons  $\sum_{j=1}^2 a_{1j}$ . Maintenant, ceci est la somme des  $a_{1j}$  avec  $j$  variant de 1 à 2, soit  $a_{11}+a_{12}$ . Une fois que la somme des  $j$  est effectuée, on refait la même chose mais avec  $i = 2$  cette fois ci. Nous aurons donc la somme  $\sum_{j=1}^2 a_{2j}$ . Cela revient à faire  $a_{21}+a_{22}$ . Finalement, nous avons :

$$\sum_{i=1}^2 \sum_{j=1}^2 a_{ij} = a_{11} + a_{12} + a_{21} + a_{22}$$

Le principe est exactement le même avec les boucles imbriquées. Tu fixes l'itération de la boucle extérieure et tu réalises la boucle intérieure en parcourant tous ses éléments. Une fois cela fait, tu vas ensuite passer à l'élément suivant dans ta boucle extérieure et tu refais la même chose.

Ceci est tout à fait modélisable avec une boucle imbriquée en Python. On considère la matrice suivante :

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

De façon générale, on note  $a_{nm}$  l'élément présent à la  $n$ -ième ligne et la  $m$ -ième colonne d'une matrice de taille  $(n,m)$ , donc une matrice qui a  $n$  lignes et  $m$  colonnes.

Modélisons ceci sur Python. Je ne détaille pas ici la construction de ce code, nous reviendrons dessus plus tard.

```
A = [
    ['a11', 'a12'],
    ['a21', 'a22']
]

sum_elements = ""

for i in range(2):
    for j in range(2):
```

```

    if j == 0 and i == 0 : #J'ai mis cette condition pour un affichage plus agréable. Il n'y aura
pas de + devant le premier terme.
        sum_elements += A[i][j]
    else :
        sum_elements += ' ' + '+' + ' ' + A[i][j]
print("La somme des éléments de la matrice est :", sum_elements)

```

La boucle te renvoie ainsi : « La somme des éléments de la matrice est :  $a_{11} + a_{12} + a_{21} + a_{22}$  ». Ici,  $i$  n'est rien d'autre que le nombre de ligne et  $j$  le nombre de colonne. On commence par sommer chaque terme de la première ligne, donc le terme de la première ligne colonne 1 et le terme en première ligne colonne 2. Ensuite, comme pour la somme double, on augmente  $i$ , on va donc à  $i = 1$ , soit la deuxième colonne (attention, dans les listes, que nous verrons après, on commence à 0 et on finit à  $n-1$ ). On va donc sommer les termes de la deuxième ligne entre eux, pour cela, on va reparcourir  $j$ .

Un autre moyen de le voir est de mettre tout cela en ligne (ce qui revient à exactement la même chose, j'ai mis cet affichage pour que ce soit plus visuel) et d'utiliser les notions relatives aux listes. Je te propose de voir cela dès que nous ferons les listes.

## LISTES

Ah bah... Quel drôle de hasard ... Nous voilà aux listes à présent ... Les listes sont primordiales. Il est impossible de réussir en Python sans maîtriser sur le bout des doigts les listes, c'est un incontournable primordial. Je t'invite alors à relire ce passage autant de fois que nécessaire.

Ici, la notion de liste est délicate. Il ne faut pas voir une liste comme on pourrait nommer une liste de course, ce n'est pas simplement une suite de données. En réalité, il s'agit d'un tableau redimensionnable. Une liste est une séquence finie d'éléments qui peuvent être de types différents.

Les listes sont notées entre crochet et tous les éléments sont séparés d'une virgule.

Comment créer une liste ?

Tu peux créer une liste en la définissant toi-même. Par exemple, voici une liste :

❖ `Liste = ['Adam', 1, True, '23', 3.14]`

Cette liste est composée de cinq éléments :

- Deux chaînes de caractères donc deux éléments de type `string` qui sont `'Adam'` et `'23'`.
- Un entier donc un élément de type `int` qui est `1`.
- Un booléen qui `True`
- Un nombre réel, donc un flottant, qui est `3.14`.

Il est important de bien noter le type de variables utilisées car cela est primordial lors d'opérations. Bien que `23` soit un nombre, ici nous l'avons défini comme une chaîne de caractère, ainsi nous ne pourrions pas l'utiliser sous cette forme pour des calculs. Nous verrons qu'on peut convertir des `int/float` en `string` par la suite.

Il est également de construire une liste via concaténation, c'est-à-dire en assemblant deux listes entre elles. Par exemple :

```
liste = ['Adam', 1, True, '23', 3.14]
liste2 = ['Le méchant', 'Star Wars', True, 0]
liste3 = liste + liste2
# liste3 = ['Adam', 1, True, '23', 3.14, 'Le méchant', 'Star Wars', True, 0]
```

Nous avons ainsi construit une liste en concaténant deux listes entre elles. Grossièrement, on peut dire que l'on a additionné ces deux listes pour n'en faire qu'une, même si additionner n'est pas grammaticalement correct.

Un autre moyen est d'utiliser un itérable. Par exemple `list(range(5))` te donnera la liste des entiers de 0 à 4 (car `range` est exclusif, donc ne prend pas le 5). On peut également l'utiliser avec des chaînes de caractères ! Par exemple, `list(« Adam »)` va te renvoyer `['A', 'd', 'a', 'm']` Ça peut être assez pratique dans certains cas.

Il y a d'autre méthode mais je ne vais pas les détailler ici car je ne suis pas sûr que tu les aies vues. Ainsi, je ne veux pas t'embrouiller l'esprit...

Maintenant que tu as ta liste, tu comprendras assez vite que l'intérêt est de pouvoir accéder aux éléments de ta liste. Mais comment faire ? Il est primordial de bien savoir le faire. Une liste est un ensemble ordonné d'éléments, c'est-à-dire que ta liste a un ordre et sauf modification de ta part, cet ordre restera inchangé. Une liste qui contient  $n$  éléments verra ses éléments être indexés par des entiers compris entre 0 et  $n-1$ .

Par exemple, avec `liste = ['Adam', 1, True, '23', 3.14]`, nous avons :

- `'Adam'` est l'élément présent à la position 0
- `1` est l'élément présent à la position 1
- `True` est l'élément présent à la position 2
- `23` est l'élément présent à la position 3
- `3.14` est l'élément présent à la position 4

Pour aller chercher un élément dans une liste, il faut simplement appeler la liste et utiliser l'indice de l'élément. Par exemple :

- `liste[0] = 'Adam'`
- `liste[1] = 1`
- `liste[2] = True`
- `liste[3] = '23'`
- `liste[4] = 3.14`

Il est également possible d'utiliser des clefs négatives ! Cela veut dire que tu partiras de la fin de la liste. Ici, les éléments iront de -1 à - $n$  ! Ce n'est plus  $n-1$  car on commence à -1 ! Voici un exemple :

- `liste[-1] = 3.14`
- `liste[-2] = '23'`
- `liste[-3] = True`
- `liste[-4] = 1`
- `liste[-5] = 'Adam'`

Cela peut être très pratique pour chercher le dernier élément de la liste ou créer une nouvelle liste qui inverse l'ordre de celle-ci.

Une notion que l'on utilisera très souvent est la longueur de la liste. On la note `len(list)`. Si tu as  $n$  éléments dans la liste, alors `len(list)` vaut  $n$ . Tu comprendras bien que, puisque la liste est indexée



de 0 à  $n-1$ , `liste[len(liste)]` est indéfini ! Nous sortons du champ de la liste... En revanche, `liste[len(liste)-1]` est tout à fait correcte et vaut le dernier élément, soit 3.14 dans notre cas.

Maintenant que cela est fait et que tu sais chercher les éléments que tu souhaites, tu peux modifier tes éléments ! Pour cela, rien de plus simple. Tu prends l'indice de l'élément que tu souhaites changer et tu lui imputes une nouvelle valeur. Par exemple, changeons le 1 dans notre liste, qui est à l'indice 1.

- `liste[1]='Le méchant'`

La nouvelle liste est maintenant : `liste = ['Adam', 'Le méchant', True, '23', 3.14]`. Ce qui est bien est que cette liste commence par une affirmation logique, mais plus encore, on peut modifier les éléments d'une liste par un type différent ! Je peux remplacer un int par un string, un float, un booléen... Et si tu veux mon avis, ça c'est méga cool...

Il y a ensuite le slicing. Le slicing est une méthode qui permet d'extraire une partie spécifique d'une liste. Cela nous permet de récupérer des sous-séquences de la liste d'origine. Le slicing fonctionne comme la fonction range, c'est de la forme suivante : `liste[ départ : arrivée : pas ]`. Le terme « départ » renvoie à l'indice du début de la tranche. Ce terme est inclusif, ainsi, si le départ est en 0, alors l'élément 0 sera présent dans la sous-séquence. Le terme « arrivée » renvoie à la dernière valeur sélectionnée. Cette valeur est exclusive. Ainsi, nous irons jusqu'à la valeur présente avant cette limite. Le pas est le pas utilisé pour parcourir la liste. Par défaut, le pas vaut 1 (donc s'il n'est pas spécifié). Supposons de la `len(liste)=k`.

- `liste[m : n]` renvoie la liste des éléments indexés allant de  $m$  à  $n-1$ .
- `liste[:n]` renvoie la liste des éléments indexés allant de 0 à  $n-1$ . Si la valeur de départ n'est pas spécifiée, elle vaut 0 par défaut.
- `liste[m : ]` renvoie la liste des éléments indexés allant de  $m$  à  $k-1$ . Si la valeur d'arrivée n'est pas spécifiée, elle vaut  $k$ , avec  $k$  la longueur de la liste.. Ainsi, le dernier indice est  $k-1$ .

Prenons la liste suivante :

- `liste=['Adam', 1, True, '23', 3.14, 'Le méchant', 'Star Wars', True, 0]`

Que vaut `liste[: 5]` ? Que vaut `liste[2 : : 2]` ? Que vaut `liste[3 : 4]` ?

- On prend les éléments ayant les indices compris entre 0 (car le départ n'est pas spécifié, donc vaut 0 par défaut) et 4 (car 5 est exclu). Nous avons donc :  
`liste[: 5] = ['Adam', 1, True, '23', 3.14]`
- On prend les éléments ayant les éléments compris entre 2 et 9-1 (9 est la longueur de la liste) car l'arrivée n'est pas spécifiée. Or, attention, nous avons un pas de deux ! Alors, nous avons la liste suivante :  
`liste[2 : : 2] = [True, 3.14, 'Star Wars', 0]`
- On prend les éléments ayant les indices compris entre 3 et 4-1, donc 3 également. Finalement, on ne sélectionne que l'élément à l'indice 3.  
`liste[3 : 4] = ['23']`

Ceci est à maîtriser obligatoirement ! Ce n'est pas spécialement évident, mais c'est nécessaire...

En fonction du type de l'élément concerné, tu peux réaliser des opérations dans ta liste. Par exemple :

- `liste[1]= liste[1] + liste[4]`  
`liste[1]` vaut maintenant 4.14  
`liste=['Adam', 4.14, True, '23', 3.14, 'Le méchant', 'Star Wars', True, 0]`

Attention, tu ne peux pas additionner un int avec un string ou un booléen. Tes variables doivent être de même type ou faire une opération entre un int et un float.

Je vais à présent te donner des méthodes sur les listes pour te faciliter la vie ! Tu dois bien connaître ces « fonctions ». Voit le comme tel, même si en réalité ce ne sont pas des fonctions. Python est un langage de programmation orienté objet, mais on ne pas rentrer dans les détails. Sache juste que tu peux utiliser une méthode sur un objet. Ici, la liste est l'objet. La syntaxe générale est la suivante : `objet.methode(parametres)`.

Voici un tableau pour expliquer tout ceci. On notera *L* notre liste.

Méthode	Description
<i>L.append(x)</i>	Ajoute <i>x</i> à la fin de la liste
<i>L.extend(T)</i>	Ajoute les éléments de <i>T</i> (une autre liste) à la fin de la liste
<i>L.insert(i, x)</i>	Ajoute l'élément <i>x</i> en position <i>i</i> de la liste, en décalant les suivants vers la droite
<i>L.remove(x)</i>	Supprime de la liste la première occurrence de <i>x</i> si <i>x</i> est présent, renvoie une erreur sinon
<i>L.pop()</i>	Supprime le dernier élément de la liste et le renvoie
<i>L.pop(i)</i>	Supprime l'élément de l'indice <i>i</i> de la liste, en décalant les suivants vers la gauche. Cette méthode renvoie l'élément supprimé
<i>L.index(x)</i>	Retourne l'indice de la première occurrence de <i>x</i> dans la liste, si <i>x</i> est présent dans celle-ci, renvoie une erreur sinon
<i>L.count(x)</i>	Retourne le nombre d'occurrence de <i>x</i> dans la liste
<i>L.sort()</i>	Trie la liste dans l'ordre croissant
<i>L.reverse()</i>	Renverse la liste

Voyons des exemples avec `liste = ['Adam', 1, True, '23', 3.14]` et `liste2 = ['Le méchant', 'Star Wars', True, 0]`

- `liste.append('Coucou')`  
`liste = ['Adam', 1, True, '23', 3.14, 'Coucou']`
- `liste.extend(liste2)`  
`liste = ['Adam', 1, True, '23', 3.14, 'Le méchant', 'Star Wars', True, 0]`
- `liste.insert(1, 'Le méchant')`  
`liste = ['Adam', 'Le méchant', 1, True, '23', 3.14, 1]`
- `liste.remove('23')`  
`liste = ['Adam', 1, True, 3.14]`
- `liste.pop()` supprime 3.14 de la liste  
`liste = ['Adam', 1, True, '23']`
- `liste.pop(2)` supprime True de la liste  
`liste = ['Adam', 1, '23', 3.14]`

- `liste.index(3.14)` renvoie 4
- `liste.count('Adam')` renvoie 1
- `liste.reverse()`  
`liste = [3.14, '23', True, 1, 'Adam']`

On ne peut pas utiliser `liste.sort()` car il faut qu'il n'y ait que des types `int` ou `string`.

Il faut faire attention avec les `True` ou `False`... On leur attribue une valeur numérique. `True` se voit attribuer 1 et `False` se voit attribuer 0. Si tu fais `liste.count(1)`, cela te renverra 2. Il y a le 1 que tu vois, et le 1 pris par le `True`. Donc c'est bon à savoir, car parfois c'est de là que peut venir une erreur.

Il y a un dernier point important qu'il faut avoir en tête car il est à l'origine de beaucoup d'erreur.

```
liste = [1, 2, 3]
liste2 = liste
liste[0] = 0
liste.append(4)

print(liste2)
```

Que va afficher le programme ? Va-t-il afficher `[1, 2, 3]` ? Eh bien non ! Le programme va afficher `[0, 2, 3, 4]`. En effet, contrairement aux variables, ici les listes vont être reliées. Si on modifie la liste, on modifie la `liste2` également. Tout est une question de mémoire ici. Lorsque nous mettons `liste = liste2`, on stocke dans la variable `liste2` la référence stockée dans `liste`. Ainsi, `liste` et `liste2` ont le même emplacement de mémoire ! Bon, c'est plus technique mais à savoir... C'est une erreur que tu feras, ou que tu as déjà faite, mais tu y passeras mdr, c'est sûr. Et c'est normal !!!! On y passe tous, il faut se tromper une fois pour ne plus l'oublier.

Le dernier point que nous allons aborder est les listes de listes. En gros, les éléments de notre liste seront des listes. C'est ainsi que nous pouvons représenter des matrices. Voici un exemple pour mieux comprendre :

- `liste = [[1,1],[2,6]]`

Cette liste est composée de deux listes. Le premier élément est la liste `[1,1]` et le deuxième élément est la liste `[2,6]`. Ainsi :

- `liste[0] = [1,1]`
- `liste[1] = [2,6]`

En revanche, tu vois bien que nous avons encore une liste, on peut alors s'amuser à aller chercher des éléments dans ces listes ! Par exemple, pour la sous-liste `[1,1]`, nous avons :

- `sous_liste1[0] = 1`
- `sous_liste1[1] = 1`

De même, nous avons pour la deuxième sous-liste :

- `sous_liste2[0] = 2`
- `sous_liste2[1] = 6`

On peut alors combiner ces conditions en une seule ligne ! Pour renvoyer deux, on doit aller dans la deuxième sous liste puis extraire le premier élément, nous avons alors :

- `liste[1][0] = 2`

La première recherche `[1]` permet d'aller chercher la bonne sous liste. Ensuite, la seconde recherche `[0]` va chercher le premier élément de la sous-liste trouvée. Dans notre cas, nous avons alors :

- `liste[0][0] = 1`
- `liste[0][1] = 1`
- `liste[1][0] = 2`
- `liste[1][1] = 6`

Avec ceci, on peut reprendre l'exemple de notre matrice ! Lorsque tu as des listes dans une liste, tu peux alors for des boucles imbriquées pour parcourir tous les éléments de tes sous-listes.

```
A = [
    ['a11', 'a12'],
    ['a21', 'a22']
]
sum_elements = ""

for i in range(2):
    for j in range(2):
        if j == 0 and i == 0 : #J'ai mis cette condition pour un affichage plus agréable. Il n'y aura
pas de + devant le premier terme.
            sum_elements += A[i][j]
        else :
            sum_elements += ' ' + '+' + ' ' + A[i][j]
print("La somme des éléments de la matrice est :", sum_elements)
```

La première boucle, dépendant de `i`, va simplement parcourir les sous-listes que tu as dans ta liste. La seconde boucle, dépendant de `j`, va quant à elle parcourir tous les éléments de tes sous-listes. Nous avons exactement la même chose !

Je pense que nous avons vu une très grosse partie de tout ce qu'il faut savoir sur les listes !

Nous allons maintenant voir les tuples et les chaînes de caractères. Je ne t'ai pas présenté les chaînes de caractères auparavant car je souhaitais te présenter les listes avant. Tu verras que c'est assez similaire dans le principe.

## Tuples

Un tuple est très semblable à une liste. La différence est que ses éléments ne sont ni modifiables et on ne peut ni ajouter ni retirer des éléments. Un tuple, tout comme une liste, est ordonné ! Le tuple est parfois préféré car il est moins coûteux en mémoire donc on y accède plus rapidement. Même si pour toi, la différence est faible, pour un ordinateur, une nanoseconde c'est immense.

Contrairement à une liste, un tuple s'écrit avec des parenthèses.

- `tuple = ('Adam', 1, True, '23', 3.14)`

*Pour chercher les éléments, tu devras utiliser les crochets, comme pour une liste (c'est le cas de toutes les recherches dans un ensemble, liste, set, chaîne de caractère, tuple, etc).*

- `tuple[0] = 1` va te renvoyer une erreur

*Il est impossible d'ajouter des éléments ou d'en enlever. Tu peux donc oublier `tuple.remove()` ou bien `tuple.append()`... Si tu désires ajouter des éléments, tu vas devoir concaténer deux tuples entre eux, ceci est autorisé. Par exemple :*

- `tuple = tuple + ('Star Wars')`  
`tuple = ('Adam', 1, True, '23', 3.14, 'Star Wars')`

*En revanche, on ne peut pas ajouter un élément à un indice précis... Techniquement, tu n'ajoutes pas d'éléments entre eux ici, tu concatènes, ce qui est différent...*

*Le reste est similaire avec les opérations sur les listes, tout est « pareil ».*

## Chaîne de caractères

*Les chaînes de caractères sont des suites de caractères quelconques. Il peut s'agir de lettre, de symboles, de chiffres, tout ce qui est possible ! Pour représenter une chaîne de caractère, il faut la mettre entre guillemets. Tu peux utiliser « » ou bien ". Le résultat est identique. Ainsi, marquer 'Adam' ou « Adam » est purement identique. Les chaînes de caractères ressemblent fortement aux tuples, qui eux-mêmes ressemblent aux listes... Il y a donc toute une suite logique et des similitudes entre eux.*

*Tu peux concaténer des chaînes de caractères entre elles. Par exemple :*

- `'Je m'appelle' + ' ' + 'Adam.'` va donner `'Je m'appelle Adam.'`

*Tu peux également récupérer la longueur de ta chaîne de caractère en utilisant `len`. Par exemple :*

- `len('Adam') = 4`

*Ceci peut être utile notamment lorsque tu réalises des messages codés. Sache que comme pour les listes ou tuples, tu peux sélectionner des éléments bien précis. Par exemple :*

- `mot = 'Pommes'`  
`mot[1] = o`  
`mot[1:5] = omme`  
`mot[:2] = Pme`

*Cela peut s'avérer très pratique dans certains cas. Essaie de voir tes chaînes de caractères comme des suites d'éléments tous collés, un petit peu comme les listes et tuples qui sont des suites d'éléments. La seule différence est que dans une chaîne de caractère, tes éléments sont tous de type `string`.*

Comme une chaîne de caractère se comporte un peu comme un tuple, tu ne peux pas ajouter ou supprimer des caractères via « remove » ou « append ». Pour ajouter des caractères à la fin de ta chaîne, tu dois nécessairement concaténer.

Tout comme les listes, les chaînes de caractères ont leurs propres méthodes. On notera s notre chaîne de caractère.

Méthodes	Description
<code>s.lower()</code>	Converti la chaîne de caractère en minuscule
<code>s.upper()</code>	Converti la chaîne de caractère en majuscule
<code>s.isupper()</code>	Vérifie si les caractères sont tous en majuscule. Renvoie True si tel est le cas, False sinon
<code>s.islower</code>	Vérifie si les caractères sont tous en minuscule. Renvoie True si tel est le cas, False sinon
<code>s.isalpha()</code>	Vérifie si les caractères sont tous des lettres. Renvoie True si tel est le cas, False sinon
<code>s.isdigit</code>	Vérifie si les caractères sont tous des chiffres. Renvoie True si tel est le cas, False sinon
<code>s.split(c)</code>	Sépare la chaîne de caractère s autour du caractère c. Le résultat est une liste de chaîne de caractère. Par exemple, «Adam le tuteur».split(« ») donne[« Adam », «le», «tuteur»]
<code>s.join(L)</code>	Il s'agit de l'inverse de l'opération split. Ici, on va réunir des chaînes de caractères grâce au caractère c. Par exemple : «-» .join(['1', '2']) donne '1- 2'
<code>s.count(c)</code>	Compte le nombre d'occurrence du caractère c dans la chaîne de caractère
<code>s.find(c)</code>	Indique la première position du caractère c dans la chaîne de caractère et - 1 s'il n'y est pas
<code>s.rjust(n)</code>	Rajoute des espaces à la fin du mot pour avoir une chaîne de caractère de longueur n. On peut utiliser <code>rjust(n, c)</code> pour remplir les espaces avec le caractère c. On peut utiliser <code>ljust(n)</code> pour remplir les espaces à gauche (r pour Right et l pour Left)
<code>s.replace(sl, s2)</code>	Remplace toutes les apparitions de la chaîne sl dans la chaîne de caractère par la chaîne s2.

Nous avons évoqué tout ce qui était à savoir sur les chaînes de caractères, du moins tout ce qui te sera utile pour ton examen et ta culture informatique...

Je vais juste revenir sur un point. Tu peux convertir des chaînes de caractères en int ou float si cette chaîne ne contient que des chiffres. Par exemple, reprenons notre ancienne liste.

```
- liste= ['Adam', 1, True, '23', 3.14]
  liste[3]=int(liste[3])
  liste= ['Adam', 1, True, 23, 3.14]
```

Inversement, tu peux convertir des int ou float en chaîne de caractères.

```
- liste= ['Adam', 1, True, '23', 3.14]
- liste[1]=str(liste[1])
- liste= ['Adam', '1', True, '23', 3.14]
```

Il est donc primordial de convertir dans un bon type avant de faire des opérations, au risque de générer une erreur. Typiquement, sans transformer '23' en int, tu ne peux pas faire  $1 + '23'$ . La bonne méthode est :

- `liste = ['Adam', 1, True, '23', 3.14]`
- `resultat = liste[1] + int(liste[3])`

Le résultat vaut alors 24.

## Fonctions

Il s'agit probablement de la dernière partie importante de ce document. Elle est très importante car il n'y a pas d'informatique sans fonctions...

Une fonction est une séquence d'instruction, dépendant de certains paramètres d'entrée, que nous appelons les arguments. La fonction doit te retourner un résultat. La fonction te permet d'effectuer un code précis que tu peux réutiliser plusieurs fois, c'est un gain de temps dans beaucoup de cas.

La syntaxe générale est la suivante :

```
def nom_fonction(argument1, ..., argumentN) :  
    #instructions  
    return valeur_de_retour
```

Ta fonction peut dépendre de plusieurs arguments. Tu dois nécessairement faire appel à tes arguments dans les instructions. Ensuite, n'oublie pas de faire un return ! Un return n'est pas un print. C'est très souvent confondu et ceci est faux. Une fonction sans return ne renvoie rien ! Imagine que tu as une fonction PGCD et que tu ne mets pas de return. Si tu fais `print(PGCD(a,b))` ça va t'afficher le PGCD de a et b. En revanche, lorsque tu vas appeler cette fonction dans un calcul, ça ne va rien te donner. Si tu fais `PGCD(a,b)/3`, ceci ne va rien te donner car `PGCD(a,b)` ne renvoie rien sans return, ça n'a donc aucune valeur.

J'insiste vraiment sur ce point car il est FONDAMENTAL. Il ne faut pas confondre return et print.

La première ligne `def nom_fonction(argument1, ... , argumentN)` est l'en-tête de la fonction. Les éléments `argument1, ... , argumentN` sont des identificateurs appelés arguments formels de la fonction et `nom_fonction` est le nom de la fonction. Pour une fonction ne prenant pas d'arguments, on écrit simplement `def fonction()`; les parenthèses étant indispensables. Le nom de la fonction est un identificateur qui suit les mêmes règles que les identificateurs de variables.

Il est important de respecter la bonne indentation. Tu dois augmenter l'indentation lorsque tu utilises une boucle while ou for, et une condition if/elif/else. A chaque fois que tu enlèves une indentation, tu sors du bloc dans lequel tu étais. Ainsi, le retour à une indentation au même niveau que « def » marque la fin de la fonction. Tout ce qui est à ce niveau ne fait plus partie de la fonction.

Avoir une fonction, c'est bien, mais savoir l'utiliser c'est mieux ! Pour l'utiliser c'est super simple, tu dois juste l'appeler. Pour ce faire, tu dois simplement marquer :

- `nom_fonction(e1, ..., eN)`

*e1, ..., eN sont des expressions que tu as choisis pour tes arguments !*

*Créons une fonction qui calcule la somme des carrés*

```
def somme_carre(n):  
    somme = 0  
    for i in range(n+1):  
        somme += i**2  
    return somme
```

*Ici, nous avons bien utilise l'argument au sein des instructions, nous avons bien un return et des indentations claires et respectées.*

```
def somme_carre(n):  
somme = 0  
for i in range(n+1):  
somme += i**2  
return somme
```

*Ceci est une aberration... Alors 'il te plaît, JAMAIS ! Même si je sais que tu le sais, parfois on se trompe d'indentation, surtout quand on a des conditions dans des conditions, dans des conditions, ou des boucles dans des boucles dans des boucles, etc... Il faut donc toujours se focaliser sur ça, car énormément d'erreur viennent de cela.*

## Fonctions pour les fichiers

*Tu n'es pas sans savoir qu'en Python, on peut utiliser print et input. En vrai, on n'utilise pas souvent les fonctions print et input directement depuis le clavier ou vers l'écran, surtout quand on a une tonne de données à traiter, comme pour une étude statistique ou un projet TIPE. Habituellement, on stocke ces données dans des fichiers spéciaux, pour pouvoir les lire et les manipuler à loisir. C'est un peu bizarre de copier-coller tout le contenu du fichier dans le script Python, donc on préfère garder les données séparées du code qui les utilise.*

*Imaginons que tu aies un script Python, et même si tes données changent, disons, à la suite de nouvelles mesures, ton script Python reste inchangé. Après avoir fait tes bidouilles magiques avec les données, tu peux même les réécrire dans un autre fichier.*

*Pour un codeur, un fichier ouvert en mode lecture, c'est un peu comme un tuyau magique d'où arrivent les données dès que le programme les demande. Et un fichier ouvert en mode écriture, c'est comme un tuyau par lequel les données du programme s'échappent. Et devine quoi ? Le clavier et l'écran, ce sont juste des tuyaux spéciaux dans cette histoire.*

*En Python, un fichier, c'est seulement une série de caractères. Quand tu ouvres un fichier, le programme se souvient où se trouve le prochain morceau d'info grâce à un petit marqueur imaginaire. Chaque fois que tu fais une opération de lecture ou d'écriture, ce marqueur se déplace un peu plus loin.*

*Voilà, c'est comme si on avait parlé de tuyaux magiques pour les fichiers !*

*Bon, c'est très formel et pas évident à visualiser, alors voyons les fonctions que tu dois connaître ainsi que leur descriptif*



<i>Fonctions</i>	<i>Description</i>
<code>f=open(nom_ du_fichier, 'r')</code>	Ouvre le fichier <code>nom_ de_fichier</code> (donné sous la forme d'une chaîne de caractères indiquant son emplacement) en lecture (r comme read). Le fichier doit exister et seule la lecture est autorisée.
<code>f=open(nom_ du_fichier, 'w')</code>	Ouvre le fichier <code>nom_ de_fichier</code> en écriture (w comme write). Si le fichier n'existe pas, il est créé, sinon il est écrasé (vidé avant utilisation).
<code>f=open(nom_ du_fichier, 'a')</code>	Ouvre le fichier <code>nom_ de_fichier</code> en ajout (a comme append). Identique au mode 'w', sauf que si le fichier existe, il n'est pas écrasé et ce qu'on écrit est ajouté à partir de la fin du fichier.
<code>f.close()</code>	Sur un fichier ouvert comme précédemment, le ferme. Cette ligne est impérative pour les fichiers ouverts en écriture, puisque le fichier n'est réellement écrit complètement qu'à la fermeture (c.f comportement de flush).
<code>f.read()</code>	Lit tout le fichier d'un coup et le renvoie sous forme de chaîne de caractères {à ne réserver qu'aux fichiers de taille raisonnable}.
<code>f.readlines()</code>	Pareil que précédemment, mais le résultat est une liste de chaînes de caractères, chaque élément correspondant à une ligne. Attention, le saut de ligne \n est présent à la fin de chaque chaîne.
<code>f.readline()</code>	Lit une unique ligne du fichier et la renvoie sous forme de chaîne (avec \n au bout). Le curseur de lecture (virtuel!) est placé en début de ligne suivante. En pratique, on sait que l'on est arrivé en fin de fichier lorsqu'un appel à cette méthode renvoie une chaîne de caractères vide.
<code>f.write(s)</code>	Écrit la chaîne s à la suite du fichier.
<code>f.writelines(T)</code>	Écrit l'ensemble des éléments de T dans le fichier f comme des lignes successives. T est une liste, une séquence, un tuple... bref, un itérable.

*Un exemple vaut mieux que mille mots, alors c'est parti !*

```
# Ouvrir le fichier en mode lecture
f = open('exemples_notes', 'r')

# Ouvrir un nouveau fichier en mode écriture
f9 = open('exemples_notes_triees', 'w')

total = 0
T = []
lignes = f.readlines()

nombre = len(lignes)

# Parcourir chaque ligne du fichier
for ligne in lignes:
```

```

# Diviser la ligne en nom et note
c = ligne.split(';')
# Ajouter le tuple (note, nom) à la liste T
T.append((int(c[1]), c[0]))
# Ajouter la note à la somme totale
total += int(c[1])

# Trier la liste par ordre décroissant de notes
T.sort(reverse=True)

# Afficher le nombre d'élèves et la moyenne
print("Le nombre d'élèves est de ", nombre, ", avec une moyenne de ", total / nombre, ".", sep="")

# Écrire les notes triées dans le nouveau fichier
for u in T:
    f9.write(u[1] + ";" + str(u[0]) + "\n")

# Fermer le fichier écriture
f9.close()

# Fermer le fichier lecture
f.close()

```

Le script procède ainsi :

- Ouverture des fichiers : Le script ouvre le fichier 'notes\_ eleves' en mode lecture ('r') et crée un nouveau fichier 'notes\_ eleves\_ trieés' en mode écriture ('w').
- Initialisation des variables : Il initialise la variable total à zéro et crée une liste vide T pour stocker les couples (note, nom).
- Lecture des lignes : Le script lit toutes les lignes du fichier 'notes\_ eleves' et les stocke dans la liste lignes.
- Traitement des lignes : Il parcourt chaque ligne dans la boucle for, utilisant la méthode split('; ') pour diviser chaque ligne en deux parties, créant ainsi le couple (note, nom). La note est convertie en entier.
- Calcul de la somme des notes : Le script ajoute chaque note à la variable totale, calculant ainsi la somme totale des notes.
- Tri de la liste : La liste T est triée dans l'ordre décroissant des notes à l'aide des méthodes sort() et reverse().
- Affichage et écriture : En fin de script, il affiche le nombre d'élèves et la moyenne à l'écran. Ensuite, il écrit dans le fichier 'notes\_ eleves\_ trieés' les lignes "nom; note" suivant l'ordre de la liste T, qui est triée par notes décroissantes.
- Fermeture des fichiers : Enfin, le script ferme les fichiers en lecture et en écriture.

# Module Numpy et Matplotlib

## Numpy

On importera ce module avec l'alias `np` : `import numpy as np`.

Construction de tableaux numpy.

- `np.zeros(n)` : crée un vecteur (ligne) à  $n$  composantes nulles.
- `np.zeros((n,m))` : crée une matrice à  $n$  lignes et  $m$  colonnes remplie de zéros; marche aussi avec des dimensions supplémentaires.
- `np.zeros(..., dtype="uint8")` : on peut préciser le type des données. Par défaut ce sont des flottants 64 bits,
- ici `uint8` signifie « entier non-signé sur 8 bits » (donc entre 0 et 255). Ce format est utile pour les images.
- `np.ones(...)` : même chose que précédemment, avec des 1 à la place des 0.
- `np.eyes(n)` : construit la matrice identité de taille  $n \times n$ .
- `np.linspace(a,b,n)` : crée un tableau à  $n$  éléments régulièrement espacés entre  $a$  et  $b$  (inclus).
- `np.arange(a,b,h)` : crée un tableau contenant les éléments  $a$ ;  $a+h$ ;  $a+2h$ ::: strictement inférieurs à  $b$ . Similaire à `range` mais avec des flottants.

Récupération de données. Avec  $M$  un tel tableau Numpy :

- `M.ndim` : nombre de dimensions de  $M$  : 1 pour un tableau « linéaire », 2 pour une matrice, etc...
- `M.shape` : tuple donnant les dimensions de  $M$ . Par exemple `(4,3)` pour une matrice  $4 \times 3$ .
- `M.size` : nombre total d'éléments : 12 pour une matrice  $4 \times 3$ .
- `M.min()`, `M.max()`, `M.sum()` parlent d'elles-mêmes. `M.min()` renvoie la valeur minimale dans  $M$ .
- `M.max()` renvoie la valeur maximale dans  $M$ . `M.sum()` renvoie la somme de tous les éléments dans  $M$ .
- argument optionnel : « l'axe » sur lequel on calcule `min`, et `max`... Par exemple si  $M$  est une matrice  $4 \times 3$ , `M.sum(0)` calcule la somme des colonnes sous la forme d'un tableau

Listes à tableaux

- `np.array(L)` : convertir une liste en tableau Numpy. Si  $L$  est une liste de listes, on obtient une matrice, etc... Là aussi on peut préciser le type.
- `M.tolist()` : opération inverse.

Cas des matrices. Avec  $M$  et  $N$  deux matrices :

- `M[i,j]`, `M[i][j]`: élément en case  $(i; j)$ .
- `M[i]`, `M[i,:]`: ligne d'indice  $i$ .
- `M[:,i]`, colonne d'indice  $i$ .
- `M[i : i+k , j : j+l]` : bloc de taille  $(k; l)$  démarrant à la case  $(i; j)$ .
- `M.copy()` : copie de la matrice.
- `M+N`, `M-N`, `M*N`, `M/N`: opérations terme à terme si  $M$  et  $N$  de même taille.
- `c*M`: multiplication de  $M$  par  $c$ .
- `M+c` : addition de  $c$  à tous les éléments de  $M$ .
- `M.dot(N)`, `np.dot(M,N)` : produit matriciel de  $M$  par  $N$  (si dimensions compatibles).
- `M.transpose()`, `np.transpose(M)` : copie transposée de  $M$ .

- `M.trace()`, `np.trace(M)` : trace de  $M$ .
- `np.concatenate((M,N),axis=0)` concatène  $M$  et  $N$  verticalement. `axis=1` pour une concaténation horizontale.

Algèbre linéaire. Le sous-module `numpy.linalg` (importé comme `import numpy.linalg as alg`) permet de faire de l'algèbre linéaire.

- `alg.det(M)` : déterminant de  $M$ .
- `alg.inv(M)` : inverse de  $M$ .
- `alg.matrix_rank(M)` : rang de  $M$ . Attention toutefois, bien souvent les coefficients seront des flottants, le résultat est à prendre avec des pincettes.
- `alg.matrix_power(M,n)` : calcule  $M^n$  pour  $n \in \mathbb{N}$
- `alg.solve(M, Y)` : résolution de  $MX = Y$ .
- `alg.eigvals(M)` : valeurs propres de  $M$  (sous forme de tableau Numpy).
- `alg.eig(M)` : valeurs et vecteurs propres de  $M$ . Les vecteurs propres sont donnés sous la forme d'une matrice de passage. Avec  $T, P = \text{alg.eig}(M)$ , on a  $M = PDP^{-1}$ , avec  $D$  la matrice diagonale dont les coefficients diagonaux sont donnés par  $T$ , si  $M$  est diagonalisable. Attention : avec des flottants, les égalités ne sont vraies qu'à  $\epsilon$  près...

## Matplotlib

On utilisera principalement le sous-module `pyplot`, qui sert à tracer des courbes. On l'importera comme suit :

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

- `plt.figure( 'titre' )` : crée une nouvelle fenêtre de tracé (vide).
- `plt.plot(X, Y)` : relie les points  $(x_i; y_i)$  par lignes brisées, les deux listes ( ou tableaux Numpy)  $X$  et  $Y$  doivent avoir même taille. On peut préciser en option :
  - la couleur : `b`, `g`, `r`, `c`, `m`, `y`, et `k` pour blue, green, red, cyan, magenta, yellow et black.
  - le style du tracé : `-` pour un trait plein, `--` pour des pointillés, `-.` pour une ligne en pointillés qui alterne
  - avec des petits points, etc...
  - les marques sur les points  $(x; y)$  : `o` pour un cercle, `v` pour un triangle vers le bas, `*` pour une étoile, `x` pour une croix, etc..

On peut également préciser une étiquette (label). Par exemple, `plt.plot(X, Y, 'y--x', label="bidule")` tracera une courbe jaune, en pointillés, avec des croix sur les points  $(x; y)$ , de nom « bidule ».

- `plt.xlim(xmin, xmax)` : fixer les bornes de l'axe des abscisses dans la figure. De même avec `plt.ylim`.
- `plt.axis([xmin, xmax, ymin, ymax])` : même chose que précédemment.
- `plt.axis('off')` : pas d'axes. `plt.axis('equal')` : axes égaux (un cercle est un cercle !)
- `plt.xlabel("nom")` : donner un nom à l'axe des abscisses. De même avec `ylabel`.
- `plt.legend(loc="upper right")` : rajout et positionnement de la légende (les étiquettes des courbes). Voir

- *l'aide pour les options.*
- *plt.show() : affichage de la fenêtre. Tant qu'on fait des plt.plot, ils sont ajoutés à la même figure, ce qui*
- *permet de tracer plusieurs courbes sur le même graphique.*
- *plt.savefig('figure.png') : sauvegarde la figure, l'extension (ici PNG) peut être précisée.*

*Voici une grande partie de la théorie nécessaire pour savoir faire ses premiers codes en Python. Bien entendu, la théorie ne vaut rien sans la pratique, il faut donc s'exercer pour progresser !*