# Programmation – introduction

## 1 Fondements de l'informatique

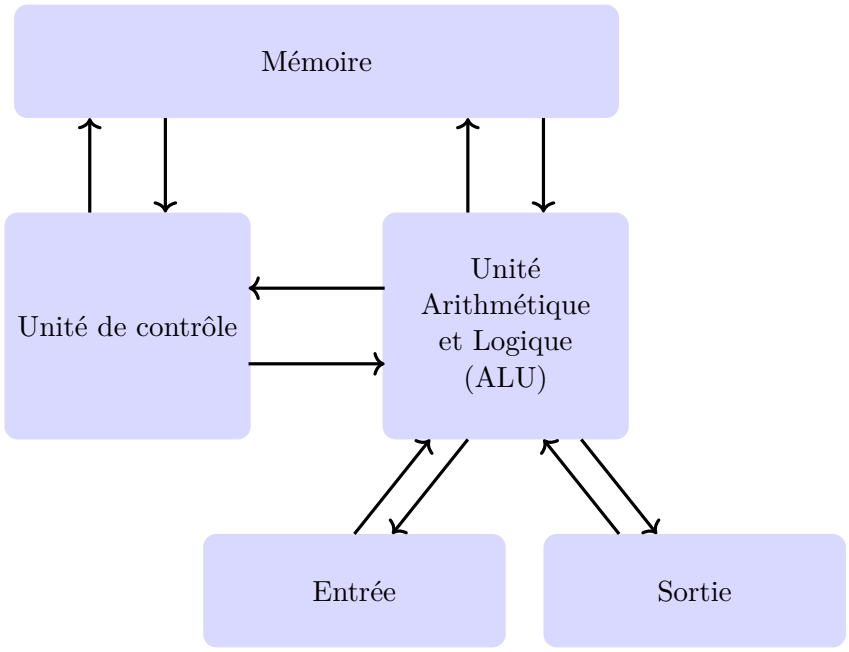
### 1.1 Que fait un ordinateur ?

Un ordinateur est une machine (de Turing) qui permet de :

- réaliser des milliards de calculs par seconde
- stocker les résultats des calculs
- télécommander d'autres machines.

Un ordinateur est rigide dans son fonctionnement : il ne connait que les informations que vous lui donnez, et il ne fait que ce que vous lui demandez.

### 1.2 Le modèle de von Neumann




Dans le modèle de von Neumann, un ordinateur est généralement composé de quatre parties :

- une (ou plusieurs) unité arithmétique et logique (ALU) qui effectue des opérations de base sur des données stockées en mémoire
- une unité de contrôle qui organise l'exécution des instructions ; elle configue le chemin de données, c'est-à-dire l'ensemble des composants dans lesquels circulent les données (ce qui inclue l'ALU et la mémoire), et organise la suite d'instructions à traiter
- la mémoire qui contient à la fois les données et les instructions ; la mémoire est constituée
  - d'une mémoire volatile (RAM pour Random Access Memory) qui contient des programmes en cours d'exécutions et et des données.

  - une mémoire morte (ROM pour Read Only Memory) dont les informations ne sont pas effacées si l'ordinateur est mis hors tension, et qui contient les programmes de base de la machine.

  La mémoire vive a l'avantage d'être plus rapide que la mémoire morte, mais elle dispose de moins de capacité que la mémoire morte, et disparait lorsque l'ordinateur est éteint.
- des dispositifs d'entrée et de sortie qui permettent de communiquer avec le monde réel (clavier, écran, micro, etc ...)

L'association de l'unité de contrôle et de l'unité arithmétique et logique est appelée processeur (CPU pour Central Processing Unit).
Notez bien que ce modèle s'est complexifié depuis son invention. Par exemple, les processeurs modernes comportent désormais plusieurs coeurs permettant de réaliser plusieurs instructions en même temps (on parle alors de programmation parallèle).

## 1.3 Abstraction et organisation des données

De façon basique, un ordinateur ne peut pas traiter n'importe quel type de données. Il peut seulement manipuler des données numériques appelés bits qui ne peuvent prendre que deux valeurs : 0 ou 1. Ces bits sont souvent regroupés en paquets de huit bits, nommés octets.
Pour manipuler des données complexes (des nombres, des mots, des images, etc ...), il est donc utile de les représenter par des suites d'octets. Ce n'est pas l'ordinateur qui donne du sens aux octets. La sémantique est la responsabilité de l'utilisateur.
De même, les opérations disponibles sur les bits et les octets sont très limitées. Cependant, Turing a montré que l'on peut tout calculer en utilisant seulement six opération de bases (appelées primitives).

Ainsi, vous devez toujours avoir à l'esprit que l'ordinateur ne prend pas d'initiative : il se contente d'appliquer bêtement ce que vous lui demandez. Bien que fondamentale en informatique, l'abstraction est votre responsabilité :

**vous devrez toujours bien réfléchir aux types d'objets que vous manipulez et au sens des commandes que vous entrez.**

Manipuler seulement des octets et quelques primitives est très fastidieux. Dans la pratique, un ordinateur se commande via des **langages de programmation**, qui fournissent des primitives additionnelles et permettent de décrire les données utilisées et les opérations qui leur seront appliquées.
Un programme appelé compilateur permet alors de traduire vos instructions en une suite de primitives applicable par l'ordinateur. De même que l'usage français requiert de suivre des règles de grammaire et d'orthographe précises :

**le compilateur ne peut faire son travail que si l'utilisateur suit des règles syntaxiques à connaitre par coeur.**

Ces règles dépendent du langage de programmation choisi. On distingue plusieurs catégories de langages de programmation appelés paradigmes. Chaque paradigme correspond à une philosophie de programmation différente.

Dans le cours qui nous intéresse, nous travaillerons du point de vue de la programmation impérative, qui est le paradigme de programmation le plus ancien. En programmation impérative, on résout un problème en listant une suite d'instructions qui modifient l'état des données jusqu'à obtenir la solution. Cette approche est aussi celle qui est privilégiée quand on suit des recettes de cuisine ou des tutoriels.

Mais il faut garder à l'esprit qu'il existe d'autres points de vue. Par exemple, en programmation fonctionnelle on considère qu'un problème se résout par évaluation de nombreuses fonctions mathématiques qui s'appellent les unes les autres. Le travail du programmeur consiste alors à diviser le problème en une collection de sous-problèmes simples à résoudre par une fonction mathématique. Cette approche sera utile pour comprendre la notion de récursivité.

## 1.4 Notions communes à tous les langages de programmation

**Définition 1.4.1** (Instruction)**.**

Une instruction est une commande de base donnée à un ordinateur. Un programme informatique est généralement constitué de plusieurs instructions.

**Définition 1.4.2** (Algorithme)**.**

Un algorithme est un ensemble d'instructions données à un ordinateur pour résoudre un problème donné.

**Définition 1.4.3** (Variable)**.**

Une variable est un nom permettant de faire référence à une donnée susceptible d'être utilisée ou transformée par un programme informatique.

**Définition 1.4.4** (Constante)**.**

Une constante est un nom utilisé pour faire référence à une valeur permanente, qui sera utilisée mais pas transformée par un programme informatique.

Toute donnée n'est qu'une suite de 0 et de 1. Pour donner du sens à ce que l'on fait, on utilise la notion de type. Le type d'une donnée est une description du type d'objet que représente cette donnée (nombre entier, nombre réel, booléen, chaîne de caractères, tableau, liste, pile, graphe, arbre, etc ...).

**Définition 1.4.5** (Type)**.**

En informatique, un type de données T est défini en donnant à l'utilisateur et/ou à l'ordinateur deux informations :

- les valeurs pouvant être prises par les objets de types $T$ ;
- les opérations autorisées sur les objets de type $T$ (appelées primitives).

*Remarque* 1.4.6.
Tous les langages n'ont pas les mêmes exigences en matière de vérification des types.

- Lorsque l'on utilise cerains langages, le type d'une variable doit être précisé lors de sa création.
- On dit qu'un langage de programmation est fortement typé lorsque la cohérence des types est garantie par le compilateur.
- D'autres langages sont dit dynamiques, au sens où ils laissent l'ordinateur attribuer les types aux variables « à la volée » lors de l'exécution du programme. Dans ce cas, l'utilisateur n'a pas besoin de préciser le type d'une variable lors de sa création (le type est alloué par Python lors de l'exécution du programme).

**Définition 1.4.7** (Déclaration)**.**

Une déclaration est un morceau de programme qui permet de renseigner l'ordinateur sur les noms et caractéristiques des éléments du programme (variables, les types, les fonctions, etc ...)

**Définition 1.4.8** (Fonctions et procédures)**.**

Les fonctions et les procédures sont de nouvelles primitives (opérations de base sur les données) définies au cours du programme par l'utilisateur.
La différence entre ces deux notions est qu'en général une fonction modifie des données et retourne un résultat, alors qu'une procédure se contente de modifier des données sans retourner de résultat.

# 2 Quelques structures de données

Dans cette section, nous décrivons quelques types disponibles lorsque l'on utilise le langage python.
En python, lorsque l'on veut connaître le type d'une variable `x` on peut utiliser la commande `type(x)`

## 2.1 Bits et octets

La mémoire d'un ordinateur permet de stocker des bits, contraction de binary digits, chiffres binaires en anglais.

**Définition 2.1.1.**

Un bit est l'élément de base que peut manipuler un ordinateur.
Un bit ne peut prendre que deux valeurs (0 ou 1) en fonction des choix de l'utilisateur.

**Définition 2.1.2.**

Un octet est une donnée constituée de 8 bits.

Comme un bit peut prendre 2 valeurs, avec $n$ bits on peut coder $2^n$ valeurs différentes. Ainsi, un octet peut prendre $2^8 = 256$ valeurs différentes.

## 2.2 Représentation des nombres

### Nombres entiers

L'implémentation des entiers en machine se fait via la représentation binaire des nombres.

**Théorème 2.2.1** (écriture binaire)**.**

*Tout nombre entier strictement positif $n$ s'écrit de manière* **unique** *sous la forme*

$$n = a_d 2^d + a_{d-1} 2^{d-1} + \cdots + a_1 2 + a_0$$

*avec $d \in \mathbb{N}$ et $a_i \in \{0; 1\}$ et $a_d = 1$. La famille de nombres*

$$a_d a_{d-1} \dots a_1 a_0$$

*est alors appelée* **écriture binaire** *de $n$.*

**Conséquence :**

Pour stocker $n$ en mémoire, il suffit de stocker les nombres $a_i$ sous forme de bits. Les nombres entiers pouvant être négatifs, on utilise un bit supplémentaire pour stocker le signe des nombres entiers.
Le nombre de bits utilisés dépend de la version de Python avec laquelle vous travaillez ! En Python 2, deux types différents permettent de représenter des nombres entiers :

- Le type `int` permet la représentation des entiers relatifs compris entre $-2^{n-1}$ et $2^{n-1} - 1$ ($n = 32$ ou 64 suivant votre plateforme) ;
  Une addition ou une multiplication peut engendrer des dépassements de capacité (le résultat peut ne pas être compris entre $-2^{n-1}$ et $2^{n-1} - 1$) ;
- Le type `long` permet la représentation des entiers avec autant de chiffres que l'on souhaite.

En Python 3, il n'y a pas de type `long` et le type `int` permet de représenter des entiers de taille illimitée.

### Nombres réels

En Python, les nombres réels sont appelés des « flottants » et sont représentés par le type `float`. En machine, le stockage des nombres réels se fait en utilisant le principe de la virgule flottante en binaire :

**Théorème 2.2.0.1** (virgule flottante en binaire)**.**

*Tout $x \in \mathbb{R}$ s'écrit de manière* **unique** *sous la forme*

$$x = \varepsilon \times 2^n \times m$$

*avec*

- $\varepsilon = \pm 1$*, le* **signe** *de $x$,*
- *et $n \in \mathbb{Z}$, l'***exposant** *de $x$,*
- $m \in [0; 1[$*, la* **mantisse** *de $x$,*

**Norme IEEE 754 en binaire :** pour stocker $x$ on stocke $\varepsilon$, $n$ et $m$.

- **Stockage du signe $\varepsilon$ :**

  il existe $r = 0$ ou $1$ tel que $\varepsilon = (-1)^r$, et on stocke la valeur de $r$ ;
  on utilise donc 1 bit pour stocker le signe ;

- **Stockage de l'exposant $n$ :**

  on suppose que $n$ prend ses valeurs entre $-1022$ et $1023$ et on stocke la représentation binaire de $e = n + 1023$ ;
  on utilise donc 11 bits pour stocker l'exposant ;

- **Stockage la mantisse $m$ :**

  on suppose que la mantisse $m$ est de la forme

  $$m = \frac{a_1}{2} + \frac{a_2}{2^2} + \cdots + \frac{a_{52}}{2^{52}}$$

  avec $a_1, \ldots, a_{52} \in \{0; 1\}$, et on stocke les valeurs de $a_1 \ldots a_{52}$ ;
  on utilise donc 52 bits pour stocker la mantisse.

*Remarque* 2.2.0.2.

1. On voit que l'on ne peut pas stocker tous les nombres réels en machine, mais seulement un nombre fini d'entre eux (ici $2^{64}$ nombres sont possibles).
   **Les données de type float ne sont que des approximations des réels !**
2. **Les nombres proches de $0$ sont considérés comme nuls.**
   Cela peut être l'origine de grosses erreurs de calculs, notamment lorsque l'on divise par un nombre très proche de 0.
3. **Un nombre peut avoir une écriture décimale finie mais une écriture binaire infinie.**
   Par exemple, $0,1$ n'est pas représentable en machine par une donnée de type float ! Lorsque l'ordinateur utilise $0,1$, il utilise en fait une approximation de $0,1$.

### Nombres complexes

Le type `complex` permet de manipuler les nombres complexes. Une donnée de type complex a une partie réelle et une partie imaginaire qui sont des flottants (type `float`).
Le nombre complexe de partie réelle $a$ et de partie imaginaire $b$ s'obtient par la commande

complex(a,b)

Si $z$ est de type `complex` alors :

- `z.real` est la partie réelle de `z` ;
- `z.imag` est la partie imaginaire de `z` ;
- `z.conjugate()` est le conjugué de `z`.

### Opération sur les nombres

| | |
|---|---|
| `-x` | l'opposé de $x$ |
| `x+y` | la somme de $x$ et $y$ |
| `x-y` | la différence de $x$ et $y$ |
| `x*y` | le produit de $x$ et $y$ |
| `x/y` | le quotient de $x$ par $y$ |
| `x//y` | la partie entière du quotient de $x$ par $y$<br>(pour des entiers, le quotient de la division euclidienne de $x$ par $y$) |
| `x%y` | la partie fractionnaire $x/y$<br>(pour des entiers, le reste de la division euclidienne de $x$ par $y$) |
| `x**y` | $x$ puissance $y$ |
| `int(x)` | la conversion de $x$ en type int |
| `long(x)` | conversion de $x$ en type long |
| `float(x)` | conversion de $x$ en type float |

### Arithmétique mixte

Python permet de faire de l'arithmétique mixte. On ordonne les types numériques Python de la manière suivante :

`int` $<$ `long` $<$ `float` $<$ `complex`.

Si le type de $y$ est plus grand que le type de $x$, alors les opérations

`x+y, x*y, x/y, x//y, x%y, ...`

sont précédées d'une conversion de $x$ en une donnée de même type que $y$.

## 2.3 Booléens

Dans un algorithme, on a souvent besoin d'exprimer des conditions ou d'évaluer la véracité d'une assertion. Pour manipuler les notions de Vrai et de Faux on utilise le type `boolean` (booléens en français).
Une variable de type booléenne ne peut prendre que deux valeurs : True et False (avec les majuscules). En terme d'implémentation informatique, les booléens sont codés avec 1 seul bit :

- 0 représente False
- 1 représente True.

Python autorise toutes les opérations mathématiques sur les booléens, et si un résultat n'appartient pas à l'ensemble $\{0,1\}$ il change le type du résultat.
Par exemple : `True` + `True` = 2 est de type `int`.

Naturellement on ajoute les opérations internes et la fonction suivantes dans l'ensemble $\{0,1\}$ :

| | |
|---|---|
| `x & y`<br>`x and y` | et |
| `x \| y`<br>`x or y` | ou |
| `x^y` | ou exclusif |
| `not x` | négation de $x$ |

Les booléens s'obtiennent souvent comme résultat des commandes suivantes :

| | |
|---|---|
| `x == y` | test d'égalité (Ne pas confondre avec l'assignation `=`) |
| `x < y` | infériorité stricte |
| `x > y` | supériorité stricte |
| `x <= y` | infériorité large |
| `x >= y` | supériorité large |
| `x != y` | différent |
| `x in y` | appartenance (listes, dictionnaires, ensembles, chaînes) |

*Remarque* 2.3.1.

La commande `not a == b` est interprétée comme `not (a == b)`, et pas comme `(not a) == b`.

## 2.4 Les types séquentiels

Les types séquentiels permettent de manipuler des familles ordonnées d'éléments (pas forcément de même type). On dispose de plusieurs types séquentiels : les listes, les tuples, les chaînes de caractères, etc ...

### Les tableaux

Un tableau est une suite d'objets de même type stockés dans des emplacements consécutifs dans la mémoire. Le nombre d'éléments d'un tableau est fixé au moment de la création du tableau et ne peut plus être modifié ensuite.

| | | | | |
|---|---|---|---|---|
| $x_0$ | $x_1$ | $x_2$ | ... | $x_n$ |

Comme les éléments sont de même type, ils occupent tous la même taille en mémoire, et l'ordinateur peut ainsi prédire le nombre de cases mémoire (bits) nécessaires pour sauvegarder le tableau. Il retrouve aussi rapidement un élément dans le tableau connaissant sa position.
Pour manipuler les tableaux on utilise le type `array`, disponible via le module `numpy`.

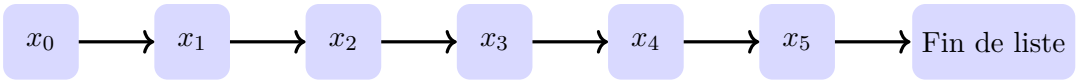
### Les listes chaînées

Parfois on veut ajouter des éléments à une famille de données à la volée, en fonction des besoins. Les tableaux sont peu adaptés à ce genre d'ajout puisqu'ils ont une taille fixée. La notion de liste permet de parer ce problème.

**Attention** : **les listes chaînées ne sont pas implémentées telles quelles en Python !**
La notion de liste en Python reprend la philosphie des listes présentée ici, mais avec une implémentation via des tableaux d'adresses en mémoire (c.f. la sous-sous-section suivante).

**Définition 2.4.0.1** (Liste chaînée)**.**

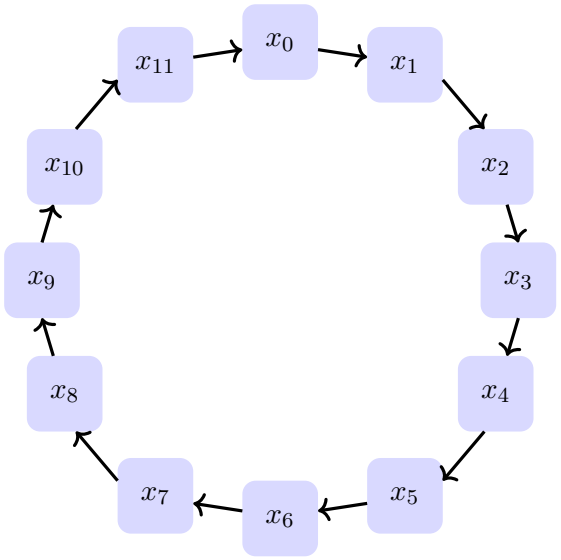
Une liste chaînée est une collection d'éléments, où chaque élément a un successeur désigné qui est soit un autre élément de la liste soit un marqueur de fin de liste.




En mémoire chaque élément $x$ d'une liste chaînée est stocké sous la forme d'un couple :

- la valeur de l'élément $x$ ;
- la position dans la mémoire du successeur de l'élément $x$

Dans un soucis de cohérence, on considère parfois que le successeur du dernier élément d'une liste est le premier élément de cette liste. On parle alors de liste circulaire :




### Les listes en Python : des tableaux dynamiques d'adresses en mémoire

En Python, les listes conservent la philosophie des listes chaînées, mais sans en reprendre l'impémentation. Ce sont encore des suites ordonnées d'éléments (pouvant être de types différents) de tailles modifiables à la volée. Par contre, en python, une liste $L$ est représentée par un tableau unidimensionel dont la i-ème case contient l'adresse en mémoire du i-ème élément de la liste $L$. Mais ce tableau ne contient pas les éléments de la liste !
Python dispose du type `tuples`, qui est similaire au type `list` à ceci près que sa taille est fixe.

**Création de listes :**

En python, les listes se manipulent à l'aide du type `list`. Pour créer une liste on dispose de plusieurs méthodes :

- pour créer la liste $L$ d'éléments $x_0, x_1, \ldots, x_n$ (dans cet ordre) on utilise la commande

  `L = [` $x_0$`,` $x_1$`, ...,` $x_n$ `]`

  La liste sans élément est appelée la liste vide et est obtenue par la commande `[]`

- pour créer une liste d'entiers consécutifs allant de $a$ à $b-1$, on peut utiliser la commande `range` :

  ```
  L = range( a, b )
  ```

  La commande `L = range( a)` est un raccourci pour `L = range(0, a)`

  construit la liste `L` dont les éléments sont les entiers compris **entre a et b-1** (inclus). De même, l'instruction

  ```
  L = range( a, b, k )
  ```

  crée la liste `L` des nombres de la forme $a + k \times i$ qui sont compris entre $a$ et $b-1$.

- on peut aussi créer une liste par compréhension : si `L` est une liste, alors la commande

  ```
  [ f(x) for x in L if condition(x) ]
  ```

  retourne la liste des valeurs des `f(x)` pour tous les `x` $\in$ `L` qui vérifient `condition(x)` (c'est-à-dire pour lesquels condition(x) vaut True).

**Accès aux éléments de la liste :**

- L'élément numéro `i` dans une liste `L` est appelé élément d'indice `i` de la liste `L`, et on l'obtient via la commande `L[i]`.

- En Python, les éléments d'une liste sont numérotés à partir de 0, pas à partir de 1.
  En particulier, `L[0]` est le premier élément de la liste `L`, et `L[i]` est le (i+1)-ème élément de la liste `L`.

- En Python, les listes sont pensées comme des listes circuclaires. Le dernier élément d'une liste `L` peut donc être obtenu par la commande `L[-1]`.
  De même l'avant-dernier élément d'une liste `L` s'obtient par la commande `L[-2]`

- **Slicing :**

  Il est possible de ne sélectionner qu'un morceau d'une liste : la commande `L[i:j]` retourne la liste des éléments de `L` d'indices compris entre `i` et `j-1`.

**Concaténation et répétition :**

- Étant données deux listes `L1` et `L2`, on peut écrire la liste `L1` suivie de la liste `L2`. On obtient ainsi une nouvelle liste, que l'on appelle **concaténation** de `L1` et `L2` la liste. Cette liste obtenue via l'instruction

  **L1 + L2** ou encore **L1.extend(L2)**

- De même, on peut contruire une liste en répétant n fois une liste `L` (c'est-à-dire en concténant `L` plusieurs fois avec elle même) via la commande

  **L * n**

  Ainsi, si `e` est un élément, alors on peut construire une liste à $n$ éléments tous égaux à `e` via la commande **[ e ] * n** .

**Autres opérations sur les listes :**

| | |
|---|---|
| `L.append (x)` | ajoute $x$ à la fin de la liste |
| `L.pop()` | retourne `L[-1]` et retire de `L` son dernier élément. |
| `x in L` | True si x est dans la liste L et False sinon |
| `x not in L` | `not ( x in L )` |
| `del L[i:j]` | enlève de la liste `L` la tranche d'éléments d'indices compris entre `i` et `j-1` (inclus) |
| `L[i:j] = t` | remplace par $t$ la tranche d'éléments de `L` d'indices compris entre `i` et `j-1` (inclus) |
| `L.count (x)` | nombre d'occurences de $x$ dans $L$ |
| `L.index (x)` | plus petit indice $i$ tel que `L[i] == x` |
| `L.insert(i,x)` | insertion de $x$ dans $L$ en position $i$ |
| `L.remove(x)` | comme `del L[ L.index(x) ]` |
| `L.reverse()` | renverse l'ordre des éléments de $L$ |

### Chaînes de caractères

En informatique, les mots et les textes sont appelés chaînes de caractères. On les manipule en utilisant le type **str** (pour string, c'est-à dire chaîne en anglais).
En python, les données de type **str** (pour string, c'est-à dire chaîne en anglais) sont des suites de caractères délimitées par **'** ou **"** (au début et à la fin de la chaîne de caractères).
En python, les chaînes de caractères se manipulent comme des listes.
En particulier, les caractères qui constituent une chaîne de caractères sont numérotées à partir de 0 et pas à partir de 1

| | |
|---|---|
| `len(X)` | longueur de la chaîne $X$ ; nombre de caractères dans la chaîne $X$ |
| `X+Y` | concaténation des chaînes $X$ et $Y$ c'est-à-dire $X$ suivie de $Y$ |
| `X[i]` | le $(i+1)$-ème caractère de la chaîne $X$ |
| `X[i:j]` | caractères d'indices $i$ à $j-1$ dans la chaîne |
| `int("23")` | conversion de "23" en l'entier 23 |

# 3 Autres notions utiles

## 3.1 Assignation

Nous avons vu qu'une variable est un nom qui permet de faire référence à une donnée stockée en mémoire. Une assignation est une opération qui attribue une valeur a une variable. Elle est réalisée via la commande

```
Var = Valeur
```

Dans cette instruction la variable est toujours à gauche du signe d'égalité, et la valeur est toujours à droite de ce signe. Ainsi, l'instruction `9 = Var` est une erreur de syntaxe.

*Remarque* 3.1.1.
Lors d'une assignation de la forme `Var = Valeur`

- si la variable `Var` n'existe pas encore, elle est crée par l'assignation et un espace mémoire est alloué au stockage de sa valeur.
- si la variable `Var` a déjà été créée, alors l'assignation modifie la mémoire elle-même pour changer la valeur de `Var`.

## 3.2 Entrées et sorties

Pour afficher un élément `x` on utilise l'instruction

```
print(x)
```

Ainsi, pour afficher le message `Hello world !` (qui est une chaîne de caractère), on peut écrire

```
print("Hello world !")
```

Ainsi, pour afficher le message `Hello world !` (qui est une chaîne de caractère), on peut écrire

```
print("Hello world !")
```

On peut aussi afficher plusieurs valeurs les unes à la suite des autres en les séparant par une virgule. Par exemple, l'instruction

```
print("La valeur de x est", 2*2)
```

affiche `La valeur de x est 4` .

Pour demander une valeur à un ordinateur et la stocker dans une variable `x`, on utilise une instruction de la forme

```
x = input("Commentaire à afficher à l'écran")
```

Ainsi la suite d'instructions suivante permet de demander à l'utilisateur son nom, puis de le saluer.

```
x = input("Entrez votre nom : ")
print("Bonjour " + x + " !")
```

Par exemple, si l'utilisateur entre Alice, l'ordinateur affiche `Bonjour Alice !` . On aurait pu obtenir le même résultat avec la suite d'instructions :

```
print("Entrez votre nom : ")
x = input()
print("Bonjour " + x + " !")
```

*Remarque* 3.2.1.

Il faut bien réaliser que input retourne une chaîne de caractères. Ainsi si on exécute le script suivant

```
a = input()
b = input()
print(a+b)
```

et que l'utilisateur entre 2 puis 1, alors l'ordinateur affiche "21" (la concaténation de "2" et "1") au lieu de 3. Pour manipuler des entiers il faut changer le type des valeurs entrées, par exemple avec la commande `int` :

```
a = input()
b = input()
print(int(a)+int(b))
```

## 3.3 Conditionnelles

Les instructions conditionnelles permettent de prendres des décisions. On les utilise pour indiquer à l'ordinateur des distinctions de cas. On peut ainsi écrire des programmes qui s'adaptent à différentes situations.
Les instructions conditionnelles les plus simples sont de la forme :

```
if <Condition> :
    instructions a faire si la condition est vraie
```

On peut aussi écrire des alternatives avec une suite d'instructions de la forme

```
if <Condition> :
    instructions a faire si la condition est vraie
else:
    instruction a faire si la condition est fausse
```

On peut aussi réaliser des distinctions de cas en utilisant une suite d'instructions de la forme

```
if <Condition1> :
    instructions a faire dans le cas ou la condition 1 est vraie
elif <Condition2> :
    instructions a faire dans le cas ou la condition 2 est vraie
elif <Condition3> :
    instructions a faire dans le cas ou la condition 3 est vraie
...
else :
    instructions a faire dans tous les autre cas
```

*Remarque* 3.3.1.

Les conditions sont des expressions booléennes pouvant être évaluées vraies ou fausses (c'est-à-dire construites à partir des opérations décrites à la section 2.3).

*Remarque* 3.3.2.

**Les indentations sont à respecter absolument en Python !** Lorsque l'on utilise une structure conditionnelle, les instructions sont écrites décalées à l'aide de quatres espaces. Cela permet à python de savoir quand s'arrêtent les instructions conditionnelles.
Ainsi, une simple erreur d'indentation peut changer complètement votre programme ! Par exemple, lorsqu'on entre la valeur -2,

le script suivant affiche -2

```
x = int ( input("entrer un nombre") )
if x > 0:
    x = x+1
    x = x**3
print x
```

Alors que le script suivant affiche - 8

```
x = int ( input("entrer un nombre") )
if x > 0:
    x = x+1
x = x**3
print x
```

Pourtant la seule différence entre les deux scripts est une simple indentation.

*Remarque* 3.3.3.

Les caractères de tabulation ne sont pas lus comme des espaces et sont donc à éviter.

## 3.4 Répétition de commande

Lorsque l'on souhaite répéter des commandes plusieurs fois on peut utiliser un type d'instructions appelé **boucle** itérative. On distingue deux type de boucles différentes :

- lorsque l'on sait combien de fois on souhaite répéter des instructions, on utilise une boucle `for` (qui siginifie "pour" en anglais)
- Si par contre on souhaite répéter une instruction jusqu'à ce qu'une condition soit satisfaite, on utilise une boucle `while` (qui signifie "tant que" en anglais)

### Boucle for

Les boucles `for` permettent de répéter une instruction un nombre connu de fois. Elles sont de la forme

```
for <variable> in <liste de valeurs> :
    instructions a effectuer pour toutes les valeurs dans la liste
```

La liste de valeur peut être construite en utilisant la fonction range. Par exemple, pour afficher les entiers de 1 à 10 et en afficher la somme, on peut utiliser le script :

```
somme=0
for a in range(1,11):
    print a
    somme=somme+a
print somme
```

Comme pour les instructions conditionnelles, **il est essentiel de faire attention aux indentations !**
Les indentations permettent de distinguer les instructions à répéter des instructions à n'effectuer qu'une seule fois.
Se tromper dans les indentations change complètement le comportement d'un programme.

### Boucle while

Parfois on veut pouvoir répéter des instructions tant qu'une condition est vérifiée (par exemple tant qu'une erreur d'approximation soit plus grande que 0.01). Dans ce cas, on ne sait pas combien de fois on va devoir répéter nos instructions. Les boucles `for` ne sont pas suffisantes pour résoudre ce type de problème. On utilise donc un autre type de structure intérative : les boucles while. Elles sont de la forme

```
while <condition> :
    instructions a effectuer tant que la condition est vraie
```

Par exemple, pour afficher un nombre carré plus grand que 123456 on peut utiliser le script :

```
i = 0
while i**2 < 123456 :
    i = i+1
print i**2
```

*Remarque* 3.4.1. **(Pensez aux initialisations !)**

Lorsque l'on utilise une boucle while, on exprime une condition d'arrêt. Toutes les grandeurs qui interviennent dans cette condition doivent avoir été données à l'ordinateur avant le début de la boucle.
Dans l'exemple ci-dessus, l'instruction `i = 0` est donc essentielle ! Sans elle votre programme ne peut pas marcher

## 3.5 Les boucles infinies.

Une boucle while ne se termine que lorsque sa condition d'arrêt est satisfaite. Si un programme fait intervenir une boucle while dont la condition d'arrêt n'est jamais vérifiée, alors se programme s'exécute indéfiniment.
Lorsque l'on propose un algorithme faisant intervenir une boucle while il faut donc vérifier la terminaison de cette boucle while, c'est-à-dire que la condition d'arrêt de cette boucle while est vérifiée en temps fini.
Pour montrer qu'un algorithme faisant intervenir une boucle while se termine, on exhibe généralement un variant de boucle.

**Définition 3.5.1** (Variant de boucle)**.**

Un variant de boucle est une grandeur $v_n$ dont la valeur dépend du nombre $n$ de répétitions des instructions de la boucle while déjà effectuées et qui vérifie toutes les conditions suivantes :

- $v_n$ ne prend que des **valeurs entières**
- **positivité :** la valeur de $v_n$ est positive
- **décroissance stricte :** pour tout $n$ on a $v_{n+1} < v_n$.

En mathématiques vous verrez le principe de récurrence finie. Ce principe a pour conséquence :

**"Si une boucle while admet un variant de boucle, alors cette boucle while se termine en temps fini"**

## 3.6 Les fonctions

### Définir des fonctions

Les opérations de bases d'un ordinateur sont souvent en nombre assez limité. Pour simplifier l'écriture de programmes informatique, la plupart des langages de programmation offre la possibilité à l'utilisateur de créer lui même de nouvelles opérations adaptées à ses besoins, que l'on appelle des routines. En python toutes les routines sont appelées fonctions. On les crée en utilisant une structure d'instruction de la forme

```
def MaFonction (x1, x2,..., xn) :
    instruction 1
    instruction 2
    ...
    instruction finale
```

Ici, `x1, x2, ..., xn` sont des variables que l'on appelle **paramètres** de la fonction. Pour utiliser la fonction il est nécessaire de donner des valeurs à chacun des paramètres.

*Exemple* 3.6.0.1.
Par exemple, pour afficher l'aire d'un rectangle on peut commencer par définir une fonction `Aire`.

```
def Aire(Longueur, largeur) :
    print(Longueur * largeur)
```

Pour afficher l'aire d'un rectangle de longueur 4 et de largeur 3, on utilise alors l'instruction

```
Aire(4,3)
```

### La commande `return`

Dans la plupart des langages de programmation, on distingue deux types de routines :

- les procédures, qui modifient les données stockées en mémoire
- les fonctions, qui réalisent des calculs et retourne un résultat.

**Cependant, cette distinction n'existe pas en Python !** Dans le langage Python, les fonctions ne se différencient des procédures que par la présence (ou non) d'une commande `return`.

Lorsqu'elle est activée, la commande `return` **interrompt** la fonction en cours et retourne une valeur qui peut alors être stockée dans une variable ou utilisée dans un calcul.

*Exemple* 3.6.0.2.

Pour tester si un nombre est premier on peut utiliser la fonction

```
def EstPremier(N):
    if N == 1 :
        return False
    if N == 2 :
        return True
    for d in range (2,N) :
        if N % d == 0 :
            return False
    return True
```

*Remarque* 3.6.0.3. (**Danger :** usage de `return` dans les boucles itératives)

Comme la commande `return` interrompt la fonction en cours et donc aussi les boucles. Il faut donc être prudent lorsque les instructions d'une boucle font intervenir une instruction de la forme `return`. Par exemple, dans le programme suivant la boucle `for` ne sert à rien :

```
def Erreur(N):
    for d in range (2,N) :
        if N % d == 0 :
            return True
        return False
```

En effet la boucle est toujours interrompue lorsque l'on traite le cas $d = 2$ et les autres cas ne sont même pas étudiés. Au final, cette fonction se contente de vérifier que $N$ est pair, ce qui aurait pu être fait de façon plus simple : la fonction suivante

```
def EstPair(N):
    return N % 2 == 0
```

teste aussi si $N$ est pair mais est beaucoup plus simple à écrire et à lire !

*Remarque* 3.6.0.4. (Différence entre `return` et `print`)

Les commandes `print` et `return` sont deux concepts très différents :

- la commande `print` est un appel à une fonction, alors que `return` est un mot-clé qui déclenche une action
- la fonction `print` affiche une valeur à l'écran et laisse le programme continuer son cours
- l'instruction return arrête **immédiatement** la fonction en cours et retourne une valeur à l'endroit même ou la fonction en cours avait été appelée

### Variables globales et variables locales

Lorsque l'on indtroduit de nouvelles variables au sein d'une fonction, ces variables n'existent que pendant l'exécution de la fonction et disparaissent ensuite. Ces variables sont dites **locales**. On les distingue des variables définies en dehors de toutes fonctions, que l'on appelle des variables **globales**.
Lorsque l'on modifie une variable globale au cours d'une fonction, cette variable reste modifiée après l'arrêt de la fonction. Les changements effectués sur les variables globales sont donc permanent !
Les variables globales peuvent être utilisées librement au cours d'une fonction. Pour pouvoir modifier une variable globale, il est nécessaire de déclarer à l'ordinateur que la variable manipulée est globale via une commande de la forme

```
global NomVariable
```

*Exemple* 3.6.0.5.

Voici un programme qui multiplie un nombre par deux puis affiche sa valeur :

```
Nombre = 2

def Double():
    global Nombre
    Nombre = Nombre *2
    print(Nombre)
```

La première fois que l'on utilise la fonction `Double`, l'ordinateur affiche 4, la seconde fois l'ordinateur affiche 8, et la $n$-ième fois l'ordinateur affiche la valeur de $2^{n+1}$.

*Remarque* 3.6.0.6.

Toutes les listes entrées comme paramètre d'une fonction sont considérées comme des variables globales.
Lorsque que vous modifiez dans une fonction `Fct` une liste `L` entrée en paramètre de cette fonction`Fct`, les modifications apportées à `L` au cours de la fonction `Fct` sont conservées après la fin de la fonction `Fct`.