

# IC1T - Programmation

Cours 1

---

## Informations générales

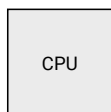
- 7h30 de **cours théoriques**, 7h30 d'**exercices à domicile**
- Les documents utilisés sont sur **claco.ecam.be**
- Evaluation :
  - Exercices à domicile (20%), **non-réévaluable**.
  - Examen écrit sur PC (80%)
- **Toutes les communications passent par Teams**

## Qu'est-ce qu'un ordinateur ?

Un ordinateur est un système composé de plusieurs éléments

Processeur, Mémoire, Stockage, périphériques, ...

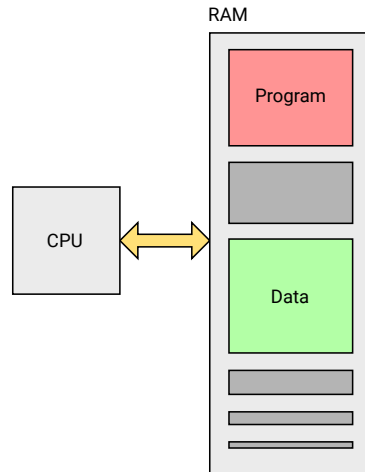
## Processeur



- Central Processing Unit (CPU)
- Exécute des instructions
- Programme = suite d'instructions
- Un programme manipule des données

Le processeur (CPU) exécute les instructions d'un programme une par une. Ces instructions proviennent de la mémoire et manipulent généralement des données se trouvant également dans la mémoire.

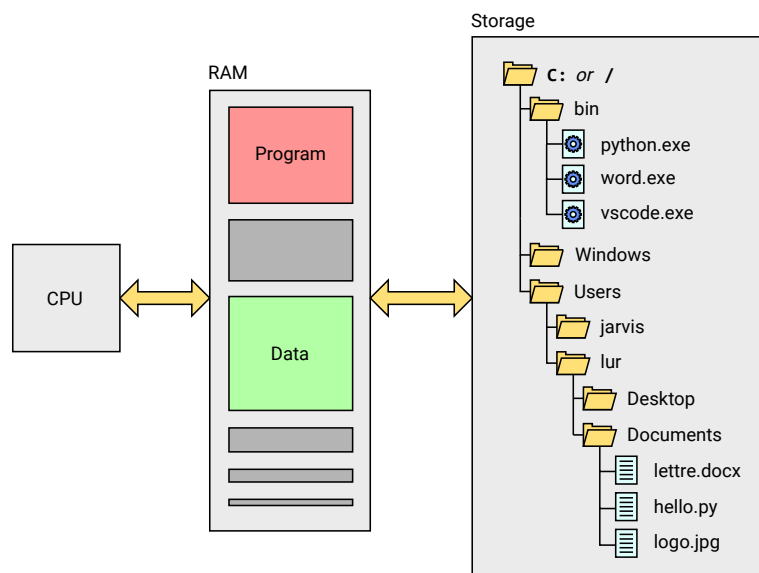
## Mémoire



- Random Access Memory (RAM)
- Contient le programme et les données utilisées par le processeur
- Rapide
- Taille limitée
- Volatile

La mémoire (RAM) contient les instructions du programme et les données qu'il manipule. Sa taille ne dépasse généralement pas quelques giga-octet (Go) et son contenu est temporaire.

## Stockage



- Solid State Drive (SSD) ou Hard Disk Drive (HDD) "disque dur"
- Contient des programmes et des données
- Lent
- Taille importante
- Persistant
- Organisé en fichiers et répertoires

Le stockage de masse (SSD, HDD, Clé USB, carte SD, ...) contient des fichiers organisés dans une hiérarchie de dossiers. Les opérations d'écriture et de lecture de fichiers sont beaucoup plus lentes que les accès à des données en RAM. Les stockages de masse n'ont pas besoin d'être alimenté pour conserver les données et peuvent contenir jusqu'à plusieurs téra-octet (To).

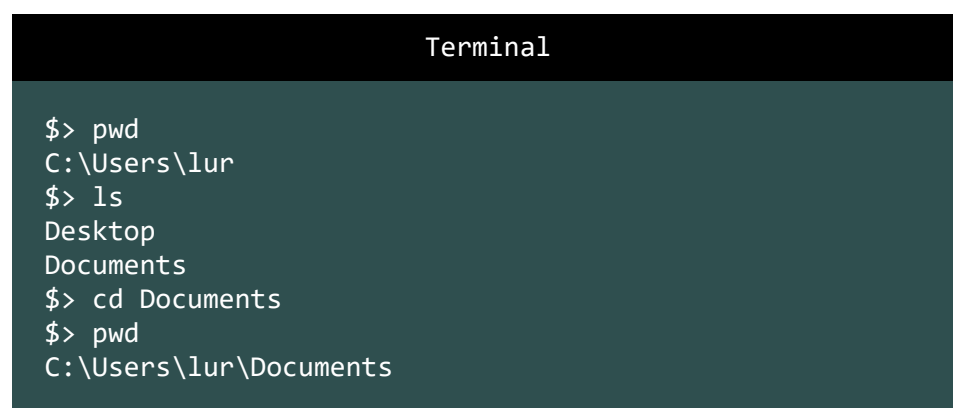
## Terminal

Une bonne partie de l'utilisation d'un ordinateur passe par la création, l'édition et la gestion de fichiers dans l'arborescence de dossiers du stockage de masse. Bien que de nos jours la plupart des gens fasse ces opérations en utilisant une interface graphique, il est aussi possible de les réaliser en ligne de commande dans un terminal.

Sous Windows, il y a plusieurs programmes permettant d'utiliser les lignes de commande :

- cmd: invite de commande
- Powershell
- Windows Terminal

Sous MacOS, le programme s'appelle simplement "terminal"



```
Terminal

$> pwd
C:\Users\lur
$> ls
Desktop
Documents
$> cd Documents
$> pwd
C:\Users\lur\Documents
```

Les développeurs et les ingénieurs sont régulièrement amenés à utiliser le terminal. Il est donc important d'apprendre les bases de son utilisation.

La première notion à intégrer est la notion de répertoire courant. Lorsqu'on utilise le terminal, il y a toujours un dossier de

l'arborescence qui est considéré comme notre position actuelle. On peut afficher le chemin de ce dossier avec la commande `pwd` pour *print working directory*. Beaucoup de commandes agissent directement sur le répertoire courant.

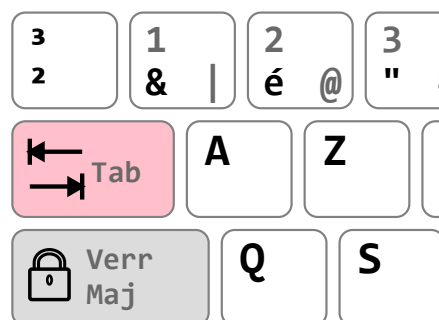
La commande `ls` permet de lister le contenu du répertoire courant

La commande `cd` (*change directory*) permet de changer le répertoire courant. On indique le nouveau répertoire courant par son chemin relatif ou absolu

- Un chemin absolu indique un éléments de l'arborescence de fichiers en partant de la racine de celle-ci :
  - Sous Windows: `C:\Users\lur`
  - Sous MacOS ou Linux: `/Users/lur`
- Un chemin relatif l'indique en partant du répertoire courant. Si le répertoire courant est `C:\Users` alors le chemin relatif `lur\Documents\hello.py` correspond au chemin absolu `C:\Users\lur\Documents\hello.py`

Remarques:

- Dans un chemin, l'utilisation de `..` permet de remonter d'un repertoire. La commande `cd ..` permet donc de passer au repertoire parent. Un `.` seul représente le repertoire courant. Le chemin relatif mentionné plus haut peut donc aussi s'écrire `..\lur\Documents\hello.py`.
- Windows utilise des anti-slash (`\`) pour séparer les différentes parties du chemin alors que MacOS et Linux utilisent des slash (`/`)
- Dans la plupart des terminaux, la touche tabulation (`Tab` ⇧) permet d'auto-compléter les éléments des chemins



## Langage machine

- Ensemble des instructions exécutables par le processeur
- Conçu pour s'exécuter rapidement
- Pas conçu pour les humains

Les fichiers contenant des programmes contiennent les instructions du programme dans un langage propre au processeur. Par exemple, le jeu d'instructions des processeurs x86 (i3, i5, i7, amd ryzen, ...) est complètement différent de celui des processeurs ARM (Apple Silicon, Snapdragon, ...). Ces jeux d'instructions sont fait pour s'exécuter efficacement mais sont particulièrement difficile à lire et à écrire par un être humain. Pour créer un programme on passe généralement par un langage de programmation.

## Langage de programmation

- Conçu pour les humains
- code source = texte brut
- Peut être compilé ou interprété

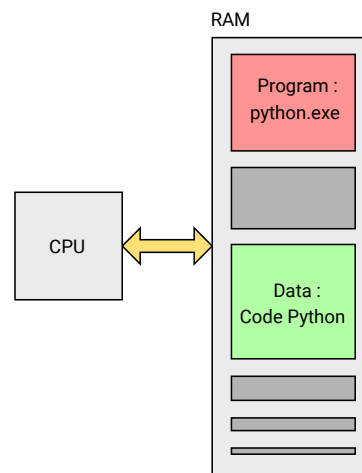
Un langage de programmation est fait pour être écrit et lu par un être humain. Le code source d'un programme est un simple fichier texte brut, c'est-à-dire qu'il ne contient que le texte du programme et pas d'information de formatage. Le code source doit ensuite être traduit en langage machine pour s'exécuter sur le processeur.

Dans un langage compilé, la traduction se fait par un programme appelé compilateur. Le compilateur génère un fichier exécutable à partir du code source. Ce fichier peut ensuite être exécuter directement sur le processeur et le compilateur n'est plus nécessaire.

Dans un langage interprété, la traduction se fait par un programme appelé interpréteur en temps réel. L'interpréteur ne produit pas de fichier exécutable. L'interpréteur est nécessaire pour chaque exécution du code source.

## Python

- Langage interprété
- L'interpréteur = `python.exe`



## Installation

- Version actuelle: 3.9.7
- Télécharger sur [python.org](https://python.org)
- Cocher la case *"Add Python 3.9 to PATH"*

Le fait d'ajouter Python à la variable PATH permet de faciliter son utilisation dans le terminal. Pour démarrer un programme à partir du terminal, il faut écrire le chemin vers son fichier exécutable. La variable PATH contient une liste de dossiers dans lesquels on peut chercher des fichiers exécutables. Si le programme qu'on souhaite démarrer se trouve dans un de ces dossiers, il suffit d'écrire son nom, le chemin n'est plus nécessaire.

## Utilisation

### Mode interactif

```
Terminal

$> C:\bin\python.exe --version
Python 3.9.7
$> python
Python 3.9.7 ...
Type "help", "copyright", "credits" or "license" for more
>>> 1 + 1
2
>>> exit()
$>
```

Ce qu'on ajoute après le chemin de l'exécutable est considéré comme des arguments. Les arguments permettent de passer des paramètres au programme que l'on démarre. Les différents arguments sont séparés par des espaces. Exemple: l'argument *--version* permet d'afficher le version de l'exécutable Python.

Le mode interactif permet d'utiliser Python comme une calculatrice. Chaque ligne de code écrite est directement évaluée et la valeur est affichée. Pour lancer le mode interactif, il suffit de démarrer Python sans argument.

### Remarque:

- sous Windows, le *.exe* peut être omis.
- Si un chemin contient des espaces, il faut le mettre entre guillemets.

## Utilisation

### Mode script

```
hello.py


print('Hello World')

Terminal


$> python hello.py
Hello World
$>
```

Le mode script permet d'exécuter le code se trouvant dans un fichier. Il suffit d'écrire le chemin du fichier en argument.

### Écrire des fichiers Python

- Texte Brut
- N'importe quel éditeur de texte
- Nous utiliserons **Visual Studio Code**
- Télécharger: [code.visualstudio.com](https://code.visualstudio.com)
- Ajouter l'extension ( icône  ) Python de Microsoft

Les fichiers sources Python peuvent être écrit avec n'importe quel éditeur de fichier texte brut. Il est cependant plus simple d'utiliser un éditeur spécialisé dans le code.

Visual Studio Code (VSCode) est un éditeur open source qui supporte plusieurs langages. Une fois VSCode installé, on peut lui ajouter des fonctionnalités en installant des extensions. L'extension Python ajoute pas mal de fonctionnalités utiles lorsqu'on développe un programme en Python. Pour installer cette extension il suffit de cliquer sur l'icône extensions () , taper "python" dans le champ de recherche et cliquer sur installer.

### Exemple de programme

```

from math import sqrt

def bazooka(a, b, c):
    delta = b**2 - 4*a*c
    if delta < 0:
        return []
    if delta == 0:
        return [ -b/(2*a) ]
    return [
        (-b - sqrt(delta))/(2*a),
        (-b + sqrt(delta))/(2*a)
    ]

print(bazooka(1, -1, -2))
# [-1.0, 2.0]

```

Voici un exemple de fichier Python. Si vous l'examinez attentivement, vous pouvez probablement deviner ce que fait ce programme.

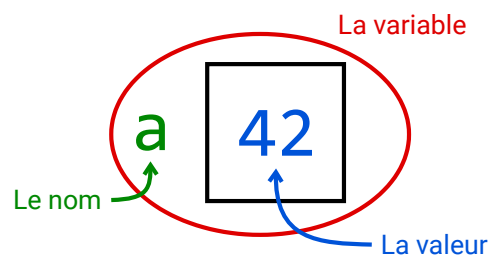
Remarque: Un # indique que la fin de la ligne est un commentaire.

Nous allons maintenant expliquer les différents éléments de ce programme.

## Valeurs et variables

- Une variable contient une valeur
- Une variable a un nom
- Le signe = sert à assigner une valeur à une variable

```
a = 42
```



Une variable est une sorte de boîte possédant un nom et pouvant contenir une valeur. Pour mettre une valeur dans la variable, on utilise le signe =. Une fois la valeur de la variable définie son contenu peut être utilisé dans le code en utilisant son nom.

```

a = 42
print(a)
# affiche 42

```



La valeur de la variable peut être modifiée en lui assignant une autre valeur.

```
a = 42
print(a) # affiche 42
a = 0
print(a) # affiche 0
```

## Types de valeurs

Les valeurs peuvent être de différents types :

- Les entiers (**int**)

```
a = 42 # littéral
a = 2 * 21 # expression
```

- Les nombre à virgule flottante (**float**)

```
a = 0.42 # littéral
a = 42e-2 # littéral en notation scientifique
a = 1 / 2 # expression
```

- Les chaînes de caractères (**string**)

```
a = "hello" # littéral
b = 2 * a + "!!" # expression
```

- Les listes (**list**)

```
a = [1, 2, 3] # littéral
b = [1, 2] + [3] # expression
```

- Les booléens (**bool**)

```
a = True # littéral
b = a or 5 < 0 # expression
```

- Et bien d'autres...

Le type d'une valeur est très important car certaines opération agissent différemment en fonction du type.

```
i = 21
print(i + i) # affiche 42
s = "21"
print(s + s) # affiche 2121
print(i + s) # plante
```

## Opérations sur les nombres

- Les opérateurs suivants sont définis pour les valeurs numériques

```
a = 1 + 2 # somme de deux nombres
a = 1 - 2 # différence de deux nombres
a = 2 * 2 # produit de deux nombres
a = 4 ** 2 # puissance de deux nombres
a = 4 / 2 # quotient de deux nombre
a = 5 // 2 # division entière
a = 5 % 2 # modulo, reste de la division entière
```

```
seconds = 265
print(seconds, 'secondes correspond à', seconds//60,
      'minutes et', seconds%60, 'secondes')
```

## Opérations sur les séquences

- Les chaînes de caractères et les listes sont des séquences.
- L'opérateur `+` entre deux séquence permet de créer une nouvelle séquence contenant les deux opérandes bout-à-bout

On appelle cette opération la concaténation

- L'opérateur `*` entre un entier `n` et une séquence permet de créer une nouvelle séquence contenant `n` fois la séquence de départ.

```
f = [1, 2] + [3, 4] # [1, 2, 3, 4]
g = 3 * "hello" # "hellohellohello"
```

## Conditions

- Le `if` permet d'exécuter un bloc seulement si une condition est remplie

```
note = -5
if note < 0:
    print("notes négatives interdite !")
    note = 0
if note > 20:
    print("notes limitée à 20 !")
    note = 20
print('note =', note)
```

- **Instruction bloc** : L'entête fini par `:` et le bloc est indenté à droite

- tabulations ou espaces pour l'indentation, **doit être identique pour toutes les lignes du bloc**

Il arrive régulièrement qu'on se retrouve avec un mélange de tabulations et d'espaces dans les indentations ce qui provoque une erreur de syntaxe et empêche le programme de se lancer. Dans VSCode, il est possible d'activer l'affichage des caractères blancs pour identifier plus facilement les espaces et les tabulations

```
if note < 0:
    print("notes")
    note = 0
```

## Écrire la condition

- Une condition est une expression **à valeur booléenne**
- **Comparaison** de deux valeurs :
  - Égalité (==) et différence (!=)
  - Strictement plus grand/petit (>, <)
  - Plus grand/petit ou égal (>=, <=)

```
a = 12 == 3 * 4 # a vaut True
b = "Eat" > "Drink" # b vaut True
c = a != b # c vaut False
```

Attention à ne pas confondre le = et le ==. = sert à assigner une valeur à une variable et == sert à tester l'égalité de deux valeurs.

## Opérateurs logique

- Opérations sur des **valeurs booléenne**
- NON logique (**not**) inverse une valeur
- ET logique (**and**) impose les deux expressions à **True**
- OU logique (**or**) nécessite une seule expression à **True**

a	b	not a	a and b	a or b
False	False	True	False	False
False	True		False	True
True	False	False	False	True
True	True		True	True

```
a = 8 > 2 and 12 <= 4 # a vaut False
b = 5 != 5 or 'PY' == 'P' + 'Y' # b vaut True
```

## Fonctions

- Qu'est-ce que c'est ?  
C'est un sous-programme réutilisable
- Comment les reconnaître ?  
Grâce aux () qui suivent son nom
- Un appel à une fonction a une valeur  
Elles peuvent être utilisées dans des expressions

```
from math import cos
from cmath import sqrt

x = cos(0)
y = sqrt(4) # racine carrée (complexe)

# on peut entrer un string
s = input("Entrez un nombre: ")
v = int(s) # conversion de valeur

# print vaut None
a = print("La réponse est " + str(42))
print(a)
```

## Nos fonctions

- On peut définir nos propres fonctions

```
def one():
    return 1
```

- Le mot `def` sert à définir une fonction
- Le mot `return` sert à indiquer la valeur de la fonction
- La définition de fonction est une instruction bloc

Il peut y avoir plusieurs `return` dans une fonction mais l'exécution du code d'une fonction s'arrête au premier `return` rencontré.

## Paramètres de fonction

- Une fonction peut prendre des paramètres

```
def somme(a, b):
    return a + b

print(somme(40, 2)) # affiche 42
```

- les valeurs de l'appel sont assignées aux paramètres

Comme s'il y avait un =