# Méthodologie de la programmation


Chapitre 1

## Introduction à la programmation (impérative)
## La programmation impérative avec Python

Pablo Rauzy <pr@up8.edu>
pablo.rauzy.name/teaching/mp

# Introduction à la programmation (impérative)

# Programmes

- Un *programme* est un ensemble d'opérations exécutables par un ordinateur.
- Quand on parle de programme, il peut s'agir de son *code source* ou d'un *binaire*.
- Le code source d'un programme peut être *interprété* ou *compilé*.

- L'interprétation d'un programme source est réalisée par un autre programme, qu'on appelle un *interpréteur*.
- Pour chaque opération décrite dans le code source, l'interpréteur :
  - lit et analyse l'opération,
  - l'évalue ou l'exécute.

\+ cycle de développement plus rapide, débuguage plus simple

− optimisation, vitesse d'exécution du programme

- La compilation d'un programme source est réalisée par un autre programme, qu'on appelle un *compilateur*.
- Le compilateur traduit le programme *une fois pour toutes* vers un autre langage.
- Généralement, vers un langage plus "proche" de l'ordinateur, pour que le code puisse être exécuté directement par le matériel.

\+ optimisation, vitesse d'exécution du programme

− cycle de développement plus lent, débuguage plus complexe

- La compilation d'un programme source est réalisée par un autre programme, qu'on appelle un *compilateur*.
- Le compilateur traduit le programme *une fois pour toutes* vers un autre langage.
- Généralement, vers un langage plus "proche" de l'ordinateur, pour que le code puisse être exécuté directement par le matériel.

\+ optimisation, vitesse d'exécution du programme

− cycle de développement plus lent, débuguage plus complexe

- Il existe des techniques intermédiaires (*bytecode*, *JIT*) qui tentent des compromis.

# Code source

- Les codes sources sont écrits dans divers langages de programmation.
- Nous allons commencer par étudier des concepts génériques des langages impératifs.

- Dans un langage impératif, un programme est une suite d'*instructions*.
- Une instruction correspond à une "commande", telle que :
  - la *déclaration* d'une *variable*,
  - l'*affectation* de la valeur d'une *expression* à une variable,
  - le test *conditionnel*,
  - l'*itération*.

- Une *variable* dans un programme a le même rôle qu'une variable en mathématique : elle symbolise (*nomme*) une valeur qui peut changer au cours du temps.
- Une variable a un *nom* et un *type*.
- Le type défini les valeurs possibles : un nombre, une chaîne de caractères, un booléen...
  - Dans certains langages le type est attaché à la variable et ne peut pas changer, on parle alors de langages *statiquement typés*.
  - Dans d'autres, le type est seulement attaché aux valeurs et donc le type d'une variable peut changer, on parle alors de langages *dynamiquement typés*.

- Une *expression* est une combinaison d'éléments du langage qui retourne une valeur quand elle est *évaluée* :
  - `2 + 3`
  - `age > 20`
  - `taille * 100`
  - `prenom + " " + nom`

- Dans certain langage, on doit déclarer une variable avant de pouvoir l'utiliser :
  - `var eleve_nom`
  - `nombre eleve_nom`
  - `chaine eleve_nom`
- Dans d'autres, il suffit d'affecter une valeur à la variable :
  - `eleve_nom` ← `"Sonia"`
  - `eleve_age` ← `20`

! Attention, dans certains langage l'affectation se fait avec le symbole `=`, sans que celui-ci n'ait de rapport avec le = des mathématiques !

- Un programme doit pouvoir faire des choix dynamiquement, lors de son exécution.
- On utilise pour cela des *tests conditionnels*.
- Ils permettent de n'exécuter un *bloc* (sous partie) du programme que si une expression booléenne est vraie :

```
si age >= 18 alors:
    afficher("majeur")
sinon:
    afficher("mineur")
```

- Un programme doit pouvoir répéter certaines opérations plusieurs fois.
- On utilise pour cela des *boucles*.
- Elles permettent d'exécuter un bloc du programme que tant qu'une expression booléenne est vraie :
  - ```
    n ← 0
    tant que n < 10 faire:
      afficher_entier(n)
      n ← n + 1
    ```
    ```

- Une *fonction* est un “sous-programme”.
- Elles permettent de réutiliser plusieurs fois le même code à différents endroits.
- Une fonction reçoit un ou des *arguments* (ou paramètres) et renvoie un résultat.
  - ```
    fonction calculer_age (annee_naissance):
      renvoyer annee_courante - annee_naissance
    ```
  - ```
    fonction fact (n):
      resultat ← 1
      tant que n > 1 faire:
        resultat ← n * resultat
        n ← n - 1
      renvoyer resultat
    ```

! Attention, la notion de fonction en programmation n’est généralement pas équivalente à celle des mathématiques.

- Pour organiser les programmes, on utilise des *structures de données*.
- Une structure de données est un *type* qui regroupe plusieurs variables.
- Il y a différent type de structures de données.
- Selon les langages, certains types de structures de données existent nativement : les listes, les vecteurs, les dictionnaires...
- On peut aussi créer ses propres structures de données.
  - Par exemple, une structure représentant un élève comprendra son prénom, son nom, son numéro d'étudiant·e, son adresse email, son UFR, son année d'étude, ses options, ...
  - En créant ainsi un type "élève" on peut utiliser une seule variable de ce type là où on a besoin de toutes ces valeurs.

# La programmation impérative avec Python

# Python

- Python a été créé en 1991 par Guido van Rossum.
- C'est un langage très répandu pour lequel il existe énormément de *bibliothèques*.
- Python est *dynamiquement typé*.
- Python est plutôt prévu pour la programmation *impérative*.
- Python offre un support de la programmation *orientée objet* (à base de classe).

# La syntaxe de Python

- La *syntaxe* de Python se veut très lisible.
- L'indentation et les retours à la ligne sont significatifs :
  - une instruction se termine avec un retour à la ligne,
  - les blocs sont marqués par l'indentation.
- Les commentaires sont tout ce qui suit le symbole # sur une ligne jusqu'à la fin de celle-ci.

# Variables, expressions, affectations

- Les variable n'ont pas de types et il n'y a pas besoin de les déclarer.
- La convention en Python est de nommer les variables en minuscules en séparant les mots par des underscores.
- L'opérateur d'affection est `=`.
- Exemples :
  - `distance = speed * duration`
  - `length_in_cm = 2.54 * length_in_inch`

! Attention, ce `=` est différent du $=$ des mathématiques !

# True, False, None

- Les valeurs booléennes en Python sont `True` et `False`.
- Il existe aussi une valeur `None` qui veut dire “rien”.
- Les opérateurs de comparaison booléenne renvoient `True` ou `False`.
- L’égalité se teste avec `==`, l’inégalité avec `!=`.
- Pour tester si une valeur `expr` est `None` on utilise `expr is None`.

# Nombres

- Les nombres peuvent être entier ($\mathbb{Z}$) ou flottant ($\sim \mathbb{R}$).
- Exemples :
  - `13.51`
  - `42`

# Chaînes de caractères

- Les chaînes de caractères sont notées entre simple ou double quote (`'` ou `"`).
- Selon lequel on utilise il faut l'échapper avec un `\`.
- Exemples :
  - `"Je m'appelle Python"`
  - `"dites \"AAAAAAH\"."`
  - `'Je m\'appelle Python'`
  - `'dites "AAAAAAH".'`
- On peut utiliser certains opérateurs sur les chaînes :
  - `"cou" * 2 # vaut "coucou"`
  - `"MIT" + "SIC" # vaut "MITSIC"`

# Tuples

- Python permet de manipuler des paires, des triplets, des quadruplets, ...
- La syntaxe est de séparer les expressions par des virgules.
- Exemples :
  - `nom, age = "Sam", 24`
  - `a, b = b, a`

! Attention `1,23` est la paire composée de `1` et `23`, pas le nombre `1.23`.

# Listes

- Une *liste* (ou *tableau*/*vecteur*, c'est confondu en Python) se notent entre crochets et leurs éléments séparés par des virgules.
- On accède à un élément d'une liste en donnant son indice entre crochets.
- Exemples :
  - `one_to_ten = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
  - `one_to_ten[3] # 4`

# Dictionnaires

- Un *dictionnaire* (ou *tableau associatif*) en Python permet de stocker des associations clef-valeur.
- On note les dictionnaires entre accolades, leurs entrées séparées par des virgules, et les clefs séparées des valeurs par des `:`.
- On accède à une valeur avec sa clef entre crochets.
- Exemple :

```
mdlp = {
  'niveau': 'L1',
  'semestre': 1,
  'enseignant·e·s': {
    'A': 'P Rauzy',
    'B': 'JJ Bourdin',
    'C': 'A Pappa',
    'D': 'S Chalençon'
  }
}

mdlp['niveau'] # 'L1'
mdlp['enseignant·e·s']['A'] # 'P Rauzy'
```

- La syntaxe des conditions est la suivante :
  - ```
    if condition:
      then-block
    elif other-condition:
      otherwise-block
    else:
      else-block
    ```
- Il peut y avoir zéro ou plusieurs bloc **`elif`** après un bloc **`if`**.
- Il peut y avoir zéro ou un bloc **`else`** à la fin.
- Les branches sont introduites par un symbole **`:`** puis délimitées par l'indentation.
- La convention en Python est d'utiliser 4 espaces comme indentation
  (j'en utilise parfois 2 par habitude, mais c'est mieux de suivre les conventions !).
    ```

- Il existe deux types de boucles :
  - “tant que”, qui répète un bloc d’instructions tant qu’une condition est vraie,
  - “pour ... dans”, qui répète un bloc d’instructions pour chaque valeur dans un conteneur.
- Leur syntaxe sont :
  - **while** *bool-expr*:
    *do-block*
  - **for** *v* **in** *iterable*:
    *do-block*
- Même syntaxe que pour les branches conditionnelles.

# Compréhension de liste

- Il y a une syntaxe spéciale pour faire des opérations sur les éléments d'une liste.
- Pour créer une nouvelle liste à partir des éléments d'une liste existante :
  - `[`*`expr(x)`* **`for`** *`x`* **`in`** *`lst`*`]`
- On peut au passage filtrer certains éléments de la liste :
  - `[`*`expr(x)`* **`for`** *`x`* **`in`** *`lst`* **`if`** *`pred(x)`*`]`
- Exemples :
  - `lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
  - `[n * 2 for n in lst]`
  - `[n * n for n in lst if (n % 2) == 0]`

- Les fonctions en Python sont définies avec le mot-clef **def**.
- La convention en Python est de nommer les fonctions en minuscules en séparant les mots par des underscores.
- On utilise **return** *expr* pour renvoyer une valeur et quitter la fonction.
  - Utilisé sans rien derrière c'est équivalent à **return** None.
- Exemples :

```
1 def fact (n):
2   result = 1
3   while n > 1:
4     result = n * result
5     n = n - 1
6   return result
```

```
1 def get_distance (speed, duration):
2   return speed * duration
```

- Appels de fonctions :
  - `fact(10)`
  - `distance = get_distance(50, 0.25)`

# Portées des variables

- On appelle la *portée* d'une variable la partie du code dans laquelle on peut y accéder.
- Les variables qui sont initialisées en dehors d'une fonction sont *globales* (accessibles partout).
- Les variables qui sont initialisées dans une fonction sont *locales*, elles n'existent que dans cette fonction.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  a = 5 # global
2
3  def fct (arg):
4    b = "foo"
5    r = b * (a + arg) # ici a est accessible
6    a = 10
7    return r
8
9  # b et r ne sont plus accessibles ici
10 # ici a vaut
```

# Portées des variables

- On appelle la *portée* d'une variable la partie du code dans laquelle on peut y accéder.
- Les variables qui sont initialisées en dehors d'une fonction sont *globales* (accessibles partout).
- Les variables qui sont initialisées dans une fonction sont *locales*, elles n'existent que dans cette fonction.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  a = 5 # global
2
3  def fct (arg):
4    b = "foo"
5    r = b * (a + arg) # ici a est accessible
6    a = 10
7    return r
8
9  # b et r ne sont plus accessibles ici
10 # ici a vaut 5
```

# Passage des arguments

- En Python les arguments sont passés aux fonctions *par référence*.
- Cela veut dire que si on fait une modification *en place* de ces variables, alors elles seront affectées en dehors de la fonction aussi.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  def f (lst1, lst2):
2    lst1 = [1, 2, 3] # affectation d'une nouvelle liste
3    lst2.append(13) # ajout d'élément en place
4
5  a = [0, 1, 0, 1]
6  b = [10, 11, 12]
7
8  print(a)
9  print(b)
10 f(a, b)
11 print(a)
12 print(b)
```

# Passage des arguments

- En Python les arguments sont passés aux fonctions *par référence*.
- Cela veut dire que si on fait une modification *en place* de ces variables, alors elles seront affectées en dehors de la fonction aussi.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  def f (lst1, lst2):
2    lst1 = [1, 2, 3] # affectation d'une nouvelle liste
3    lst2.append(13) # ajout d'élément en place
4
5  a = [0, 1, 0, 1]
6  b = [10, 11, 12]
7
8  print(a) # [0, 1, 0, 1]
9  print(b)
10 f(a, b)
11 print(a)
12 print(b)
```

# Passage des arguments

- En Python les arguments sont passés aux fonctions *par référence*.
- Cela veut dire que si on fait une modification *en place* de ces variables, alors elles seront affectées en dehors de la fonction aussi.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  def f (lst1, lst2):
2    lst1 = [1, 2, 3] # affectation d'une nouvelle liste
3    lst2.append(13) # ajout d'élément en place
4
5  a = [0, 1, 0, 1]
6  b = [10, 11, 12]
7
8  print(a) # [0, 1, 0, 1]
9  print(b) # [10, 11, 12]
10 f(a, b)
11 print(a)
12 print(b)
```

# Passage des arguments

- En Python les arguments sont passés aux fonctions *par référence*.
- Cela veut dire que si on fait une modification *en place* de ces variables, alors elles seront affectées en dehors de la fonction aussi.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  def f (lst1, lst2):
2    lst1 = [1, 2, 3] # affectation d'une nouvelle liste
3    lst2.append(13) # ajout d'élément en place
4
5  a = [0, 1, 0, 1]
6  b = [10, 11, 12]
7
8  print(a) # [0, 1, 0, 1]
9  print(b) # [10, 11, 12]
10 f(a, b)
11 print(a) # [0, 1, 0, 1]
12 print(b)
```

# Passage des arguments

- En Python les arguments sont passés aux fonctions *par référence*.
- Cela veut dire que si on fait une modification *en place* de ces variables, alors elles seront affectées en dehors de la fonction aussi.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  def f (lst1, lst2):
2    lst1 = [1, 2, 3] # affectation d'une nouvelle liste
3    lst2.append(13) # ajout d'élément en place
4
5  a = [0, 1, 0, 1]
6  b = [10, 11, 12]
7
8  print(a) # [0, 1, 0, 1]
9  print(b) # [10, 11, 12]
10 f(a, b)
11 print(a) # [0, 1, 0, 1]
12 print(b) # [10, 11, 12, 13]
```

# Arguments par défaut

- Il est possible de spécifier une valeur par défaut pour certains arguments (qui doivent tous être après ceux qui n'ont pas de valeur par défaut).
- Exemple :

```
1 def get_distance (speed, duration = 1/6):
2   # par défaut, on calcul la distance parcouru en 10 minutes}
3   return speed * duration
4
5 get_distance(50, 0.5) # une demi-heure en agglomération
6 get_distance(130)     # 10 minutes sur l'autoroute
```

1. Écrire un programme qui compte de 0 à 100, mais remplace les nombres divisibles par 3 par “Fizz”, ceux divisibles par 5 par “Buzz”, et ceux qui le sont par 3 et par 5 par “Fizz Buzz”.
2. Écrire une fonction qui prend une liste de contacts et une lettre et renvoie la liste des emails de celleux dont le nom commence par cette lettre (on imagine que chaque contact est enregistré comme dictionnaire avec comme informations son prénom, son nom, et son adresse email).

▶ Informations (peut-être) utiles :

- La fonction `print(a)` affiche la valeur de son argument *a*.
- `range(`*min*`,`*max*`)` retourne une liste dont les éléments sont les nombres de *min* à *max* (non-inclus).
- Les caractères d'une chaîne de caractères sont accessibles comme les éléments d'une liste.
- La fonction `str.lower(s)` renvoie la chaîne *s* mais avec toutes les lettres en minuscules.