



Méthodologie de la programmation

Chapitre 1

Programmation impérative avec python

1. Notions de programmation

2. Programmation impérative avec Python

Variables

- ▶ 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 définit 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**.

Déclaration et affectation

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

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

Expressions

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

Conditions

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 :

```
1 si age >= 18 alors :  
2     afficher ( "majeur" )  
3 sinon :  
4     afficher ( "mineur" )
```

Boucles

Un programme doit pouvoir **répéter certaines opérations** plusieurs fois.

- ▶ On utilise pour cela des boucles.
- ▶ Elles permettent de d'exécuter un bloc du programme que tant qu'une expression booléenne est vraie :

```
1 n ← 0
2 tant que n < 10 faire :
3     afficher_entier(n)
4     n ← n + 1
```

Fonctions

≠ des fonctions en mathématiques

- ▶ 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.

```
1 fonction calculer_age (annee_naissance):  
2     renvoyer annee_courante – annee_naissance
```

```
1 fonction fact (n):  
2     resultat ← 1  
3     tant que n > 1 faire :  
4         resultat ← n * resultat  
5         n ← n – 1  
6     renvoyer resultat
```


Structures de données

Pour organiser les programmes, on utilise des **structures de données**.

- ▶ Il y a différents **types** 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 lorsqu'on a besoin de toutes ces valeurs.

1. Notions de programmation

2. Programmation impérative avec Python

Python, un langage impératif

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

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'affectation est `=`.
 - `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** (tableau associatif) stocke 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.

```
1 mdlp = {  
2     'niveau': 'L1',  
3     'semestre': 1,  
4     'enseignant.e.s': {  
5         'A': 'JP Palus',  
6         'B': 'A Millour'  
7     }  
8 }  
9 mdlp['niveau'] # 'L1'  
10 mdlp['enseignant.e.s']['A'] # 'P Rauzy'
```

Conditions

- ▶ La syntaxe des conditions est la suivante :

```
1 if condition:
2     then-block
3 elif other-condition:
4     otherwise-block
5 else:
6     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

Boucles

Deux types de boucles

- ▶ “tant que”, qui répète un bloc d'instructions tant qu'une condition est vraie,

```
1 while bool-expr:
2     do-block
```

- ▶ “pour ... dans”, qui répète un bloc d'instructions pour chaque valeur dans un conteneur.

```
1 for v in iterable:
2     do-block
```

Compréhension de liste

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 :

```
1 [expr(x) for x in lst]
```

- ▶ On peut au passage filtrer certains éléments de la liste :

```
1 [expr(x) for x in lst if pred(x)]
```

- ▶ Exemples :

```
1 lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2 [n * 2 for n in lst]
3 [n * n for n in lst if (n % 2) == 0]
```

Fonctions

Les **fonctions** en Python sont définies avec le mot-clef **def**.

- ▶ convention : nommer les fonctions en minuscules en séparant les mots par des underscores.
- ▶ On utilise `return expr` pour renvoyer une valeur et quitter la fonction. (`return` seul équivaut à `return None`).

```

1 def fact (n):
2     result = 1
3     while n > 1:
4         result = n * result
5         n = n - 1
6     return result
7
8 def get_distance (speed, duration):
9     return speed * duration
  
```

- ▶ `fact(10)`
- ▶ `distance = get_distance(50, 0.25)`

Portée d'une variable

= **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.

```

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**.

- ▶ 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.

```
1 def f(lst1, lst2):  
2     lst1 = [1,2,3]  
3     lst2.append(13)  
4 a = [0,1,0,1]  
5 b = [10,11,12]  
6 print(a)  
7 print(b)  
8 f(a, b)  
9 print(a)  
10 print(b)
```

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).

```
1 def get_distance (speed, duration = 1/6):  
2     return speed * duration  
3  
4 get_distance(50, 0.5) # une demi-heure en agglomeration  
5 get_distance(130) # 10 minutes sur l'autoroute
```

TD

- ▶ Écrire un programme qui affiche les nombres de 0 à 100, mais remplace les nombres divisibles par 3 par « Trois », ceux divisibles par 5 par « Cinq », et ceux qui le sont par 3 et par 5 par « Trois et Cinq ».
- ▶ Écrire une fonction retournant la liste des n premiers nombres de Fibonacci sachant que :
 - $F_0 = 0$
 - $F_1 = 1$
 - $F_n = F_{n-1} + F_{n-2}$

Sources

- ▶ Cours de Pablo Rauzy ([lien](#))
- ▶ Cours de Jean-Pascal Palus ([lien](#))