# Programmation : vocabulaire et bonnes pratiques

| Type | Valeur | Itérable | Séquence | Mutable |
|---|---|---|---|---|
| int | entier | non | non | non |
| bool | `True` ou `False` | non | non | non |
| float | décimal | non | non | non |
| str | chaine de caractères | oui | oui | non |
| tuple | tableau d'adresses | oui | oui | non |
| list | tableau d'adresses | oui | oui | oui |
| dict | tableau d'adresses | oui | non | oui |

## Objectifs

A la fin de la séquence d'enseignement les élèves doivent :

- reconnaitre des **données**, des **variables**, des **opérateurs**, des **expressions**, des **instructions** et des **fonctions**.
- déterminer le portée lexicale de variables
- être conscients des effets de bord
- comprendre l'intérêt et comment documenter une fonction ou un programme
- distinguer les types d'erreurs
- savoir comment les prévenir

## Table des matières

# 1 Introduction à la programmation

## 1.1 Introduction

### 1.1.1 Structure de l'informatique

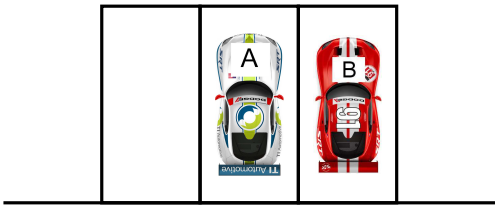
L'informatique est structurée par quatre concepts :

- **l'algorithme :** méthode opérationnelle permettant de résoudre un problème,
- **la machine :** système physique doté de fonctionnalités,
- **le langage :** moyen de communication entre l'informaticien et la machine,
- **l'information :** données symboliques susceptibles d'être traitées par une machine.

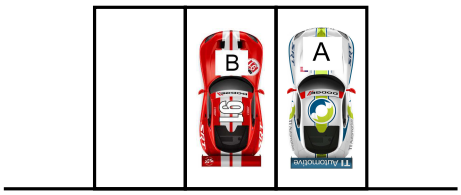
On se propose ici d'acquérir une méthodologie permettant de mettre sur le papier la résolution d'un problème bien posé. Pour cela, nous utiliserons un ensemble d'instructions dont l'application permet de résoudre le problème en un nombre fini d'opérations.

### 1.1.2 Poser proprement le problème

**Problème:** permuter deux voitures sur un parking de trois places numérotées $P_1$, $P_2$ et $P_3$, sans gêner la circulation. La voiture A est sur l'emplacement $P_2$, la voiture B est sur l'emplacement $P_3$.



Avant permutation



Après permutation

Un premier algorithme *naïf* peut être le suivant :

1. déplacer la voiture B de l'emplacement $P_3$ à $P_1$
2. déplacer la voiture A de l'emplacement $P_2$ à $P_3$
3. déplacer la voiture B de l'emplacement $P_1$ à $P_2$

Plusieurs imprévus peuvent se présenter :

- l'emplacement P1 est-il initialement libre ?
- les voitures A et B sont-elles en état de marche ?
- y-a-t-il de la circulation pendant l'échange des emplacements ?
- l'emplacement $P_3$ est-il pris pendant l'échange ?

Cela est dû à un problème en partie mal posé. Précisons les choses :

- **Données :**
  Parking de trois emplacements, numérotés $P_1$, $P_2$ et $P_3$.
  Deux voitures en état de marche : A sur $P_3$, et B sur $P_2$.

- **Hypothèses générales :** Un seul conducteur réalise la permutation.
  Celui-ci a son permis de conduire et les clés des deux voitures.
  Lorsqu'une voiture est en mouvement, aucune autre voiture ne vient sur le parking.
  Une éventuelle autre voiture ne reste qu'un temps fini sur un emplacement.

- **Résultats :** La voiture B est sur l'emplacement $P_2$.
  La voiture A est sur l'emplacement $P_3$.

L'algorithme est alors le suivant :

1. aller au volant de la voiture B (emplacement $P_3$)
2. **répéter** attendre **jusqu'à** emplacement $P_1$ libre
3. déplacer la voiture B de l'emplacement $P_3$ à $P_1$
4. aller au volant de la voiture A (emplacement $P_2$)
5. **répéter** attendre **jusqu'à** emplacement $P_3$ libre
6. déplacer la voiture A de l'emplacement $P_2$ à $P_3$
7. aller au volant de la voiture B (emplacement $P_1$)
8. **répéter** attendre **jusqu'à** emplacement $P_2$ libre
9. déplacer la voiture B de l'emplacement $P_1$ à $P_2$

### 1.1.3 Première approche méthodologique

Un *bon* algorithme doit :

- être clair
- fournir un résultat satisfaisant
- être adapté au public visé

Pour cela, il est nécessaire de :

- poser correctement le problème
- rechercher une méthode de résolution
- écrire l'algorithme par raffinements successifs

## 1.2 Pseudo-langage algorithmique

Afin d'uniformiser l'écriture des algorithmes, un pseudo-langage (non normalisé) contenant l'indispensable est utilisé : des **données**, des **variables**, des **opérateurs**, des **expressions**, des **instructions** et des **fonctions**.

### 1.2.1 Données et constantes

Une donnée est une valeur extérieure introduite dans l'algorithme. Une constante est une valeur non modifiable (0, `True`, $\pi$...).

### 1.2.2 Variables et affectation

Une variable est un objet dont la valeur est modifiable. Elle possède, via une affectation :

1. un identificateur : nom (suffisamment explicite)
2. un type : simple (booléen, entier, réel, caractère) ou structuré (chaîne de caractères, liste, tuple, dictionnaire, ...),
3. une valeur
4. une référence : indique l'adresse de stockage dans la zone mémoire.

Si `var` est la variable et `val` sa valeur, l'affectation se fait en écrivant `var = val` en langage **Python** . En pseudo-code, on écrira : `var` ← `val`.

### 1.2.3 Opérateurs

- Opérateurs arithmétiques

| Nom | Symbole |
|---|---|
| addition | + |
| soustraction | - |
| produit | * |
| division | / |
| division entière | div ou // |
| modulo | mod ou % |
| exposant | ^ ou ** |

- Opérateurs relationnels

| Nom | Symbole |
|---|---|
| identique | == |
| différent | <> |
| inférieur | < |
| supérieur | > |
| inférieur ou égal | <= |
| supérieur ou égal | >= |

- Opérateurs logiques

| | |
|---|---|
| négation | non, ~, ¬ |
| ou | ou, —, ∨ |
| et | et, &, ∧ |
| opération d'affectation | ← |
| Concaténation de liste | + |

Chaque type de donnée admet un jeu d'opérateurs adapté.

### 1.2.4 Expressions

Une expression est un groupe d'opérandes (nombres, constantes, variables,...) liées par des opérateurs. Elles sont évaluées pour donner un résultat.

**Exemple :** expression numérique : $b/a$ et expression booléenne : $a \neq 0$ (vrai ou faux).

### 1.2.5 Instructions

Une instruction est une commande exécutée par la machine. Elle peut contenir une ou plusieurs expressions.

Il existe des instructions **simples** comme l'affectation $x \leftarrow b/a$ ou l'appel de fonction : `print("x=", x)`.

Mais il y a aussi des instructions **structurées** : groupe d'instructions, structures conditionnelles et structures itératives (répétitives).

#### 1.2.5.1 Groupe d'instructions simples

**Algorithm 1** Résolution de l'équation du $1^{er}$ degré $a.x = b$

**entrée:** $a$, un réel non nul et $b$ : un réel
**résultat:** x
1: $x \leftarrow b/a$
**renvoi:** Affiche("x = ",x)

#### 1.2.5.2 Instructions composées

Les instructions composées se terminent en **Python** par : et une indentation lors du retour à la ligne.

- Structures de contrôles
  - boucles conditionnelles
  - boucles inconditionnelles
- Définition d'une fonction, d'une classe...

### 1.2.6 Fonctions

Les fonctions permettent :

- d'automatiser des tâches répétitives
- d'ajouter de la clarté à un algorithme
- d'utiliser des portions de code dans un autre algorithme

Le pseudo-langage algorithmique possède comme tout langage informatique des **fonctions primitives** d'entrée/sortie telles que **Entrer**, **Afficher**, mais aussi des fonctions primitives mathématiques (**cos**, **racine**, ...). Il est aussi possible d'écrire ses propres fonctions.

Pour cela, il faut tout d'abord répondre à ces deux questions :

- quelles sont les données ?
- quel doit-être le résultat ?

Il faudra ensuite commenter l'algorithme : usage de la fonction, ...

## 1.3 Pour une *bonne* programmation

### 1.3.1 Effets de bord

En **Python** , l'affectation est une instruction. En **C**, l'affectation est un opérateur. Ainsi, le code suivant autorisé en **C**, conduit à une erreur de syntaxe en **Python** :

```
>>> k = 3
>>> i = (j = k) + 1
SyntaxError: invalid syntax
```

C'est un choix de conception du langage (cf Guido Van Rossum concepteur de **Python** ).

Cela a des conséquences sur la portée des variables. On dit qu'une fonction à un effet de bord (ou *side effect*) si son exécution modifie un état en dehors de son environnement local.

```
a, b, L = 3, 2, [1, 2]

def portee(a):
    L.append(3)
    a = 5 + b
    return a
```

Le code de gauche conduit à l'interprétation donnée à droite.
La variable `a` de type `int` n'a pas été modifiée en dehors de la fonction quand la variable `L` de type `list`, donc mutable, a été modifiée.

```
>>> print(a, b, L)
3 2 [1, 2]
>>> print(portee(7))
7
>>> print(a, b, L)
3 2 [1, 2, 3]
```

```
a, b, L = 3, 2, [1, 2]

def portee(a):
    global b
    L.append(3)
    a = 5 + b
    b = 10
    return a
```

Le code de gauche conduit à l'interprétation donnée à droite.

Sans l'instruction `global b`, l'interprétation renverrait une erreur. Bien qu'étant une variable de même type que `a`, `b` a été modifié par la fonction.

```
>>> print(a, b, L)
3 2 [1, 2]
>>> print(portee(7))
7
>>> print(a, b, L)
3 10 [1, 2, 3]
```

```
UnboundLocalError: local variable 'b' referenced before assignment
```

Lorsqu'une expression fait référence, dans une fonction, à une variable, **Python** regarde d'abord si elle est définie à l'intérieur de la fonction et sinon dans le programme principal. C'est ce qu'on appelle la portée d'une variable ou portée lexicale.

Une bonne pratique consiste à choisir des arguments de fonction différents des noms des variables du programme principal. C'est ce qu'on appelle principe de localité.

Cependant, l'effet de bord associé aux listes a permis d'écrire les tris insertion, bulle, sélection et rapide *sur place*. Cela est rendu possible en conservant la liste à la même adresse avec la méthode `append` ou l'opérateur `+=`.

```
def ajout(L, a):
    L.append(a)
```

```
def ajout(L, a):
    L += [a]
```

```
def ajout(L, a):
    L = L + [a]
    return L

L = ajout(L, a)
```

Trois façon d'ajouter un élément à `L`. Seuls les deux premières fonctions laissent `L` à la même adresse.

### 1.3.2 Rappels sur les types

On rappelle que :

- un itérable est un objet qui peut renvoyer un à un ses éléments (comme une chaine de caractère, un tuple, une liste, un ensemble)
- une séquence est un objet qui permet d'accéder à ses éléments par leur indice (et non leur clé comme pour un dictionnaire).

| Type | Valeur | Itérable | Séquence | Mutable |
|---|---|---|---|---|
| int | entier | non | non | non |
| bool | `True` ou `False` | non | non | non |
| float | décimal | non | non | non |
| str | chaine de caractères | oui | oui | non |
| tuple | tableau d'adresses | oui | oui | non |
| list | tableau d'adresses | oui | oui | oui |
| dict | tableau d'adresses | oui | non | oui |

### 1.3.3 Décomposer une problème en sous problèmes

Dans l'industrie, il arrive que des équipes produisent des codes de millions de lignes, dont certains mettent en jeux des vies humaines (aéronautique, nucléaire, ...).

Toute la difficulté réside alors dans l'écriture de programmes sûrs. On constate que lorsqu'un algorithme est simple, le risque d'erreur est moindre. C'est sur ce principe essentiel que se basent les *bonnes* méthodes.

**Ainsi, tout problème compliqué doit être découpé en sous-problèmes simples**

Chaque module ou sous-programme est alors testé et validé séparément. Tous les modules doivent aussi être largement documentés.

# 2 Pratiques de programmation

**Remarque :** Merci aux collègues de l'UPSTI pour cette partie.

## 2.1 La documentation

### 2.1.1 Spécifications de fonctions

La documentation d'une fonction est essentielle à sa bonne utilisation. En python, elle est définie en début de fonction, sous forme d'un *Docstring*, délimitée par triple-guillemets en début et en fin de documentation et indentée comme le reste de la fonction.

L'autre façon de décrire proprement une fonction consiste à établir ses spécifications, les deux éléments suivants sont à définir :

- les *préconditions* sont les conditions qui doivent être satisfaites sur les paramètres d'entrées et l'état global du programme, avant l'appel de la fonction. Il est important de préciser le **type** des variables attendues ;
- les *postconditions* sont les conditions qui seront satisfaites sur la valeur de retour et sur l'état global du programme, après l'appel de la fonction, si les préconditions ont été respectées, là aussi il est bien de préciser le **type** des variables renvoyées en sortie.

Une spécification est donc un *contrat* entre celui qui implémente une fonction et celui qui l'utilise. L'utilisateur s'engage à respecter les préconditions avant d'appeler la fonction et le programmeur lui garantit que les post-conditions seront satisfaites en retour.

La **signature** d'une fonction est l'ensemble des paramètres (avec leurs noms, positions, valeurs par défaut et annotations), ainsi que l'annotation de retour d'une fonction. C'est-à-dire toutes les informations décrites à droite du nom de fonction lors d'une définition.

```
def addition(a:int, b:int) -> int:
    """ Renvoie la somme de 2 nombres entiers a et b
    ex: addition(3, 5) -> 8
    """
    return a + b

# la signature de la fonction addition est donc ici
"(a:int, b:int) -> int"
```

Pour **appeler la documentation** d'une fonction, il est possible d'utiliser le `help` ou le `?`.

### 2.1.2 Annotations des programmes

Des **commentaires** doivent toujours figurer dans les programmes : ils permettront aux autres programmeurs de comprendre ce que l'auteur des lignes a voulu faire. Ils aussi utiles pour relire ses propres programmes. Les commentaires sont introduits par un caractère # (tout ce qui suit le # sur la même ligne est un commentaire). Ils peuvent aussi figurer entre des triple-guillemets (`"""` suivis de `"""`).

**En programmation, on essaie de dire : 1 commentaire par ligne ou par bloc d'instruction !**

## 2.2 Gestion des erreurs

### 2.2.1 Types d'erreurs

On peut distinguer trois types d'erreurs :

- une erreur **de syntaxe** survient lorsque le code source du programme est mal formé. Une telle erreur se produit, par exemple, lorsqu'on oublie la condition d'un `if`, lorsqu'on a un `else` sans `if` associé, lorsqu'on a mal écrit un mot réservé, ....
- une erreur **d'exécution** survient lorsqu'un programme, syntaxiquement correct, effectue une opération interdite. Une telle erreur se produit, par exemple, lorsqu'on tente de faire une division par zéro, lorsqu'on ajoute une liste dans un ensemble, lorsqu'on tente d'accéder à une variable d'instance privée hors de la classe, ....
- une erreur **logique** survient lorsqu'un programme, sans erreur de syntaxe ni d'exécution, ne produit pas le résultat correct attendu. Par exemple, si on a écrit `length + width` au lieu de `length * width` pour calculer la surface d'un rectangle, le programme se terminera sans erreur, mais avec une réponse erronée.

### 2.2.2 Assertion

On peut vouloir vérifier que des conditions qui sont censées être satisfaites le sont effectivement, à l'aide du mécanisme d'assertion proposé par **Python** . Par exemple, pour vérifier les préconditions de la fonction `pourcentage` :

```
def pourcentage(score, total):
    assert total > 0, 'total doit etre strictement positif'
    assert 0 <= score, 'score doit etre positif'
    assert score <= total, 'score doit etre inferieur a total'
    return score / total * 100
```

Trois instructions `assert` ont été utilisées pour vérifier les préconditions. Une telle instruction se compose d'une condition (une expression booléenne) éventuellement suivie d'une virgule et d'une phrase en langue naturelle, *sous forme d'une chaine de caractères*.

L'instruction `assert` teste si sa condition est satisfaite. Si c'est le cas, elle ne fait rien et sinon elle arrête immédiatement l'exécution du programme en affichant éventuellement la phrase qui lui est associée.

```
>>> print(pourcentage(15, 20), '%')
75.0 %
>>> print(pourcentage(25, 20), '%')
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    print(pourcentage(25, 20), '%')
  File ".\INFO-MPCSI-Cours-Programmation.py", line 4, in pourcentage
    assert score <= total, 'score doit etre inferieur a total'
AssertionError: score doit etre inferieur a total
```

Ce mode de programmation, qui exploite les assertions pour vérifier les préconditions, est appelé programmation défensive. Dans ce type de programmation, on suppose que les fonctions sont appelées comme il faut, dans le respect de leurs préconditions. On prévoit néanmoins un garde-fou avec des instructions `assert` pour vérifier que les préconditions sont effectivement bien satisfaites.

### 2.2.3 Mécanisme d'exception : try-except

Le mécanisme d'exception, présent dans les langages de programmation orientés objet, permet de gérer des exécutions exceptionnelles qui ne se produisent qu'en cas d'erreur.

La construction de base à utiliser est l'instruction `try-except` qui se compose de deux blocs de code. On place le code "risqué" dans le bloc `try` et le code à exécuter en cas d'erreur dans le bloc `except`.

```
a = input('Annee de naissance ? ')
try:
    print('Tu as', 2023 - int(a), 'ans.')
except:
    print('Erreur, veuillez entrer un nombre.')
print('Fin du programme.')
```

- si l'utilisateur entre un nombre entier, l'exécution se passe sans erreur et son âge est calculé et affiché :

```
Annee de naissance ? 2003
Tu as 20 ans.
Fin du programme.
```

- si l'utilisateur entre une chaine de caractères quelconque, qui ne représente pas un nombre entier, un message d'erreur est affiché :

```
Annee de naissance ? préhistoire
Erreur, veuillez entrer un nombre.
Fin du programme.
```

Chaque type d'erreur est défini par une *classe spécifique*. On va pouvoir associer plusieurs blocs `except` à un même bloc `try`, pour exécuter un code différent en fonction de l'erreur capturée. Lorsqu'une erreur se produit, les blocs `except` sont parcourus l'un après l'autre, du premier au dernier, jusqu'à en trouver un qui corresponde à l'erreur capturée. Exemple en capturant les exceptions spécifiques pour deux cas d'erreur (erreur de conversion et division par zéro) :

```
try:
    a = int(input('a ? '))
    b = int(input('b ? '))
    print(a, '/', b, '=', a / b)
except ValueError:
    print('Erreur de conversion.')
except ZeroDivisionError:
    print('Division par zero.')
except:
    print('Autre erreur.')
```

```
try:
    a = int(input('a ? '))
    b = int(input('b ? '))
    print(a, '/', b, '=', a / b)
except Exception:
    print('Autre erreur.')
except ValueError:
    print('Erreur de conversion.')
except ZeroDivisionError:
    print('Division par zero.')
```

L'ordre des blocs `except` est très important et il faut les classer du plus spécifique au plus général, celui par défaut devant venir en dernier. En effet, si on commence par un bloc `except` pour une exception de type *Exception*, il sera toujours exécuté et tous les autres qui le suivent ne le seront jamais. Dans l'exemple suivant, ce sera donc toujours « `Autre erreur.` » qui sera affiché dès qu'une erreur se produit dans le bloc `try`.