# Chapitre 3
# Structure du langage Python

## 1. L'interpréteur Python

### 1.1 Terminologie

Si vous lisez ce livre, c'est que vous avez un objectif : découvrir la programmation, créer un jeu, apprendre le langage Python, créer une petite application, faire des exercices ou acquérir des connaissances supplémentaires !

Tous ces objectifs vont passer par l'écriture de lignes de code constituant le **code source** de votre programme. Ces lignes de code sont stockées dans des fichiers texte ayant l'extension py. Ces fichiers sont modifiables dans un éditeur de texte, comme Notepad ou WordPad sous Windows ou encore TextEdit sous Mac ou encore avec un éditeur tel que gedit, Kate, nano ou vim sous Linux.

Le code source au format texte n'est pas directement exécutable par l'ordinateur, il doit passer à travers une étape de conversion effectuée par l'**interpréteur Python**, qui a la tâche de lire le code source et de l'exécuter.

Le langage Python dispose de sa propre grammaire interne avec ses règles et ses usages. Par exemple, le langage définit une liste de **mots-clés** réservés qui ne peuvent être choisis par le programmeur pour créer des noms de variables ou des noms de fonctions. Vous connaissez par exemples les mots-clés correspondant aux instructions `for`, `if`, `then` et `else`. Il existe aussi d'autres mots-clés comme ceux associés aux opérateurs logiques `or`, `and` et `not` ou ceux correspondant aux constantes comme `True` et `False`.

Les curieux peuvent consulter la liste des mots-clés dans la documentation officielle du langage Python à l'adresse :
https://docs.python.org/fr/3/reference/lexical_analysis.html

Lorsque l'interpréteur Python parcourt le code source, il catégorise chaque élément rencontré en utilisant les règles grammaticales du langage. Pour vous aider à saisir comment l'interpréteur se comporte, voici quelques exemples :

– Si l'interpréteur rencontre le signe #, il considère que le texte placé après ce caractère représente un commentaire du programmeur.
– La présence d'un guillemet lui permet de détecter le début d'une chaîne de caractères comme dans l'écriture "`Bonjour`".
– Si l'interpréteur rencontre un groupe de caractères commençant par une lettre, il va en premier lieu vérifier si ce terme appartient à la liste des mots-clés du langage (`for`, `and`, `if`...) et, dans ce cas, l'interpréteur en déduit son rôle : une boucle, un test, une condition... Si le terme identifié n'est pas un mot-clé, l'interpréteur sait alors qu'il s'agit d'un **nom de variable** ou d'un **nom de fonction** créé par le programmeur.
– Si l'interpréteur rencontre un groupe de caractères commençant par un chiffre, il en déduit que ce groupe correspond à un nombre comme 123 pour un entier, 0.23 pour un flottant, 0b010011 pour un nombre binaire ou encore 0xF1A2 pour un nombre hexadécimal.

## 1.2 Les éléments du langage

Une fois que l'interpréteur a analysé l'ensemble du code source, chaque élément présent dans le code a été associé à un rôle précis. Nous pouvons citer les plus courants :

– Un **littéral** (*literal* en anglais) : ce terme technique correspond à la notion de donnée écrite « en toutes lettres » dans le code source, comme une chaîne de caractères `"Bonjour"`, un nombre (123) ou encore un booléen (`True`).
– Un **opérateur** (*operator* en anglais) comme les opérateurs arithmétiques : + - / * ou les opérateurs booléens : `or`, `and`, < ...
– Un **nom de fonction**.
– Un **nom de variable**.
– Une **instruction** du langage, comme une boucle `for` ou un branchement conditionnel `if`.

**Remarque**

*Comme vous pouvez le remarquer, nous ne citons plus les commentaires. En effet, ceux-ci sont présents dans le code source du programme et sont utiles aux programmeurs, mais ils n'existent plus dans le programme en cours d'exécution car ils ne déclenchent pas de traitement effectué par le processeur.*

Dans le langage Python, les fonctions sont disponibles de base sans que l'on ait de manipulation particulière à faire. Ces fonctions préinstallées par défaut s'appellent des **fonctions natives**. On peut citer par exemple les fonctions `min()`, `max()` ou `len()`. D'autres fonctions existent dans les librairies Python comme sinus et cosinus par exemple, mais pour être utilisables dans le code, il faut importer la librairie correspondante comme ceci :

```
import math                 # la fonction cosinus est
a = math.cos(math.pi / 2)   # disponible depuis la
b = math.cos(math.pi / 2)   # librairie math
c = min(a,b)                # la fonction min est
                            # accessible de base
```

Nous avons cité des exemples assez simples, mais le travail effectué par l'interpréteur est très complexe. En effet, il existe des situations beaucoup plus ambiguës. Par exemple, une paire de parenthèses peut avoir différents rôles : elle peut correspondre à l'appel d'une fonction `Dessiner-Cercle(...)`, elle peut délimiter une partie dans un calcul `(4*3+2)` ou encore servir à définir un tuple `(4,5,6)`, cela dépend du contexte !

## 1.3 Exercices

### Exercice 1

Déterminer le rôle de chaque élément en gras. Les réponses possibles sont : littéral, opérateur, fonction, variable, instruction :

```
1.  for i in range(4) :
2.  a = b + 4
3.  Région = "Océanie"
4.  a = 4 + test() + 6
5.  if a < 4 :
6.  t = t + 8
7.  a = a + calcul()
8.  TraceCercle(x,y,r)
```

Réponses : 1. Instruction, 2. Variable, 3. Littéral (chaîne de caractères), 4. Fonction, 5. Opérateur (booléen), 6. Littéral (numérique), 7. Opérateur (arithmétique), 8. Variable

### Exercice 2

Les propositions suivantes sont-elles correctes ou incorrectes ?

1 - Le programmeur peut ajouter des mots-clés au langage.

2 - Pour utiliser la fonction `sinus()`, il suffit d'écrire son nom dans le code.

3 - On peut commencer un nom de variable par un chiffre.

4 - Les parenthèses servent uniquement pour les fonctions.

5 - Les littéraux dans un programme correspondent uniquement aux valeurs numériques écrites directement dans le programme comme 123 ou 12.34.

**Correction :**

1 - Non, c'est interdit, les mots-clés sont définis dans la norme du langage. On ne peut les modifier, en ajouter ou en retirer.

2 - Non, car `sinus()` n'est pas une fonction native, il faut importer la librairie associée : math.

3 - Faux, cela est réservé à l'écriture des nombres.

4 - Pas uniquement, elles servent aussi à écrire des calculs ou à créer des tuples.

5 - Pas uniquement, les littéraux correspondent aux autres valeurs écrites « en toutes lettres » par le programmeur dans le code source. Cela inclut aussi les chaînes de caractères.

## 1.4 Pourquoi présenter cette terminologie ?

Bien que fondamental, le vocabulaire présenté est-il un passage obligé pour savoir programmer ? La réponse est évidemment non. On peut effectivement se lancer tête baissée dans le codage sans avoir acquis cette terminologie. Si l'apprentissage par essai/recherche peut avoir un côté motivant, il nécessite cependant beaucoup plus de de temps. En effet, on peut directement se lancer dans l'écriture de quelques lignes de code en une minute, mais il faudra souvent passer plusieurs minutes à corriger les erreurs. Pourtant, l'interpréteur Python produit des messages d'erreur censés décrire l'origine des problèmes, mais les débutants n'arriveront pas à en tirer quelque chose pour les raisons suivantes :

– Les messages d'erreur sont en anglais. Cela est dommageable et cette situation ne va pas évoluer dans un futur proche.
– Les messages d'erreur utilisent le vocabulaire que nous avons présenté. L'interpréteur Python ne connaît pas votre niveau en programmation. Il s'exprime comme si vous étiez un programmeur expérimenté.

Face à cette situation, plusieurs stratégies s'offrent à vous :

– Rester dans un périmètre très délimité pour éviter toute erreur. Pour cela, faites des programmes de quelques lignes ou recopiez des exemples. En cas d'erreur, mieux vaut abandonner et passer à un autre exemple/exercice.
– Tenter de corriger les erreurs. Cependant, vous ne savez pas trop comment faire. Vous effacez alors la partie de code posant problème et la réécrivez autrement en espérant que cela supprimera l'erreur.
– Essayer de comprendre le message d'erreur de l'interpréteur et corriger le code en conséquence pour qu'il se comporte comme souhaité à l'origine.

Vous l'aurez compris : si vous ne voulez pas avancer à tâtons, mieux vaut comprendre le message fourni par l'interpréteur lorsqu'il détecte une erreur.

## 1.5 Analyse des messages d'erreur

Si vous tenez à tester les exemples suivants, il suffit de lancer l'interpréteur Python en mode interactif en tapant dans une console `python` ou `python3`.

**Exemple 1 :**

```
>>> a = b
NameError: name 'b' is not defined
```

Dans cette ligne, l'interpréteur doit initialiser la variable `a` à partir d'une variable `b`. La variable `a` peut ne pas exister préalablement et elle sera alors créée à ce niveau. Cependant, l'interpréteur a besoin de la variable `b` et elle n'a jamais été déclarée. Il va alors indiquer que le nom de la variable `b` n'a jamais été défini (not defined) ce qui sous-entend que le nom de la variable `b` lui est inconnu.

**Exemple 2 :**

```
>>> toto()
NameError: name 'toto' is not defined
```

Cette ligne ressemble à un appel de fonction. L'interpréteur aurait pu signaler que cette fonction était inconnue, mais il préfère, ne sachant pas si `toto` correspond à une variable ou à une fonction, indiquer que le nom `toto` est inconnu (not defined).

**Exemple 3 :**

```
>>> a = 5R + 1
SyntaxError: invalid decimal literal
```

L'erreur vient de l'écriture `5R` qui, suivant les conventions du langage ne correspond à rien. En effet, ce terme commençant par un chiffre, l'interpréteur considère qu'il ne peut pas correspondre à un nom. Il choisit alors de le traiter comme un littéral numérique (decimal literal). Ensuite, comme cette écriture est incorrecte, il la qualifie d'invalide (invalid).

**Exemple 4 :**

```
a = true
NameError: name 'true' is not defined
```

Peut-être aimeriez-vous que l'interpréteur signale qu'une majuscule a été oubliée ? Mais il ne donne pas de conseils, malheureusement. L'interpréteur parcourt des listes de règles pour analyser le code, et lorsque son raisonnement aboutit à une impasse, il produit alors un message expliquant pourquoi il se retrouve bloqué. Bref, pour comprendre son message, il faut retrouver le fil de son raisonnement à lui ! Ainsi, dans cet exemple, le terme `true` ne correspond pas à un mot-clé du langage. L'interpréteur en déduit donc qu'il s'agit soit d'un nom de variable, soit d'un nom de fonction. Or, ce nom est inconnu, ce qui lui pose problème.

**Exemple 5 :**

```
>>> toto = 4
>>> toto()
TypeError: 'int' object is not callable
```

Ce cas peut vous sembler simple car les deux lignes à l'origine de l'erreur se trouvent côte à côte. Cependant, dans la réalité, ce sera rarement le cas. Dans cet exemple, le nom `toto` correspond à une variable entière. Or, en écrivant `toto()`, on s'en sert comme d'un nom de fonction, d'où le problème. Ainsi, l'interpréteur vous signale qu'un nombre entier (int object) ne peut être appelé (not callable) comme une fonction.

**Exemple 6 :**

```
>>> a = b +
SyntaxError: incomplete input
```

Ce message d'erreur n'est pas très clair. Dans la logique de l'interpréteur, l'opérateur + prend deux arguments disposés à sa droite et à sa gauche. Dans cet exemple, nous avons oublié le terme de droite. L'interpréteur nous informe donc qu'il manque des informations en entrée (input) de cette opération.

**Exemple 7 :**

```
a = 4 + 3 )
SyntaxError: unmatched ')'
```

Dans cet exemple, nous avons oublié une parenthèse ouvrante. L'interprétateur signale que la parenthèse fermante reste sans correspondance (*unmatched*).

**Exemple 8 :**

```
>>> "toto" = 4
SyntaxError: cannot assign to literal here.
```

Sur la gauche se trouve une chaîne de caractères, donc un littéral. On peut attribuer (assign en anglais) une valeur à une variable, mais pas à une chaîne de caractères (literal). C'est exactement ce que signifie ce message.

**Exemple 9 :**

```
>>> True = 0
SyntaxError: cannot assign to True
```

L'interpréteur détecte qu'un mot-clé réservé `True` est utilisé dans une affectation comme un nom de variable. Alerte rouge, ce n'est pas possible ! Le message rappelle juste cette interdiction sans préciser qu'il s'agit d'un mot réservé.

# 2. Les variables

## 2.1 Usage

Une variable sert à représenter une valeur par un nom. Vous retrouvez ici un concept assez proche de ce que vous connaissez en mathématiques ou en physique. Cependant, il y a quelques différences :

– En informatique, une variable est toujours connue, car elle est associée à une case mémoire qui contient sa valeur. Le concept d'inconnue n'existe pas !

– En informatique, une variable évolue : sa valeur change au cours du temps suivant les actions effectuées durant l'exécution du programme. Lorsque l'on modifie une variable, la nouvelle valeur écrase la précédente et l'ancienne valeur est oubliée. En mathématiques et en physique, les inconnues ont une unique valeur !

Un nom de variable doit respecter certaines règles :

1. Un nom ne doit pas commencer par un chiffre.

2. Un nom ne doit pas contenir de caractères spéciaux : ! % @ # [ + = \ ( ) [ ] = & : . , ; $..., exception faite de l'underscore _ présent sur la touche [8] du clavier.

Aujourd'hui, un projet informatique peut être mené par des personnes de nationalités différentes à travers le monde. Par conséquent, l'usage veut que l'on utilise les 26 lettres standards et que l'on évite les caractères spéciaux issus de sa propre langue comme é, à ou ç pour le français. Cependant, sachez que l'interpréteur Python accepte ces caractères si vous vouliez les utiliser.