Chapitre 1 :
Programmation en Python

- **Complément sur les opérations**

Le symbole // renvoie le quotient de la division euclidienne :

`4 // 2` renvoie 2 `4 // 3` renvoie 1 `4 // 5` renvoie 0

Le symbole % renvoie le reste de la division euclidienne (modulo) :

`10 % 5` renvoie 0 `10 % 3` renvoie 1

- **Complément sur les affectations**

`a , b = 1,2` équivaut à `a = 1` et `b = 2`

`x += y` équivaut à `x = x + y`

- **L’indentation**

L'indentation, décalage vers la droite du début de ligne, est un élément de syntaxe important en Python. Elle délimite des blocs de code et elle aide à la lisibilité en permettant d'identifier facilement ces blocs. La ligne précédant l'indentation se termine par le signe “ **:** ” (deux-points).

3. Instructions conditionnelles et boucles

• Instructions conditionnelles

La structure la plus simple est le “**if**” :

```
if condition :
    instructions
```

Le mot `condition` désigne une expression et le mot `instructions` désigne une instruction ou un bloc d’instructions écrites sur plusieurs lignes à effectuer lorsque la condition est vraie.

```
if n % 2 == 0 :
    n = n // 2
```

La structure “**if-else**” ajoute une alternative pour executer une autre instruction ou bloc d’instructions à effectuer lorsque la condition est fausse :

```
if condition :
   instructions1
else :
   instructions2
```

Par exemple :

```
if n % 2 == 0 :
   n = n // 2
else :
   n = 3 * n + 1
```

La structure “**if-elif-else**” ajoute plusieurs alternatives conditionnelles :

```
if condition1 :
   instructions1
elif condition2 :
   instructions2
elif condition3 :
   instructions3
...
else :
   instructions4
```

Par exemple :

```
if n % 4 == 0 :
   n = n // 4
elif n % 4 == 1 :
   n = ( 3 * n + 1 ) // 4
elif n % 4 == 2 :
   n = n // 2
else :
   n = ( 3 * n + 1 ) // 2
```

Remarques :

- Les mots `if` et `elif` sont toujours suivis par une expression qui prendra la valeur `True` ou `False`, la ligne se termine par le symbole “ **:** ” (deux points)
- Le mot `else` est toujours immédiatement suivi par le symbole “ **:** ” (deux points)
- C’est l’indentation qui permet de délimiter les blocs d’instructions à exécuter lorsque les conditions sont vérifiées

Note :

Les mots `True`, `False` et `None` sont les rares mots en Python dont l’écriture commence par une lettre majuscule.

- **Boucles conditionnelles**

La structure est la suivante :

```
while condition :
   instructions
```

La structure est identique à celle du “`if`”. Le bloc d’instructions indenté qui suit est executé tant que la condition est vérifiée. C’est ce bloc d’instructions qui doit modifier la valeur de la condition pour qu’elle ne soit plus vérifiée et que le programme sorte du bloc et continue son exécution.

```
while a  >= b :
   a = a - b
   n = n + 1
```

- **Boucles non conditionnelles**

Une boucle non conditionnelle sera répetée n fois, n étant connu à l'avance, la structure est la suivante :

```
for i in range(n) :
    instructions
```

Ce code est équivalent à :

```
i = 0
while i < n :
    instructions
    i = i + 1
```

Dans les deux cas, une variable `i` est créée et prendra successivement les valeurs 0,1,2…,n-1 pour la boucle `for` et 0,1,2,...,n pour la boucle while et les instructions seront executées `n` fois.

La boucle “`for`” en Python propose d’autres possibilités, si on considère `objet` comme une séquence de plusieurs éléments (chaine de caractères, liste…) on pourra utiliser :

```
for element in objet :
    instructions
```

Exemple :

```
for car in “Bonjour” :
    print(3 * car)
```

Lorsqu'on utilise la fonction range avec des entiers :

`for i in range(f)` : i prendra la valeur des entiers successifs de 0 inclus à f exclu si f > 0

`for i in range(d,f)` : i prendra la valeur des entiers successifs de d inclus à f exclu si f > d

`for i in range(d,f,p)` : i prendra la valeur des de d inclus à f exclu avec un pas de p.

Attention, erreur classique :

```
for x in liste :
    x = 2*x
```

Ce code ne modifiera pas la liste existante.

Remarques :

L'instruction `break` permet d'interrompre une boucle

L'instruction `continue` permet d'éviter un passage dans la boucle

```
for i in range(10):
    if i == 3 :
        continue
    if i == 8 :
        break
    print(i)
```

Renverra : `0, 1, 2, 4, 5, 6, 7`

4. Les fonctions

- **Définition d'une fonction**

Une fonction se définit de la manière suivante :

```
def nomDeLaFonction(argument1,argument2):
    """ aide sur la fonction : docstring
    (facultatif) """
    corpsDeLaFonction
```

`def` est un mot clé du langage Python, les arguments sont séparés de virgules.

Le corps de la fonction est un bloc de code indenté.

Si le corps de la fonction contient l'instruction `return`, alors l'appel de la fonction est une **expression** qui a donc une valeur.

S'il n'y a pas d'instruction return dans le corps de la fonction, alors l'appel de la fonction renverra la valeur "None". Ce type de fonction s'appelle une **procédure**.

Exemple :

```
def volume(longueur,largeur,hauteur):
    """ renvoie le volume d'un parallélépipède
    dont on fournit les trois dimensions en
arguments."""
    return longueur * largeur * hauteur
```

- **Espace et portée des variables**

L’espace local d’une fonction contient les paramètres qui lui sont passés, et les variables définies au sein de celle-ci.

Pour qu’une variable soit utilisée dans une fonction, il faut que la variable appartienne à son espace local ou à l’espace qui appelle la fonction.

Une fonction ne peut pas modifier, par affectation, la valeur d'une variable extérieure à son espace local.

```
x=3
def f(x):
    x+=2
    print(x)

f(x) # affiche 5
print(x) # affiche 3
```

La variable `x` utilisée dans la fonction est distincte de la variable `x` définie au début du programme (`x=3`) et n'existe plus après l'appel de la fonction.

Après l'instruction `f(x)`, l'espace local de la fonction f est détruit.

Il existe un moyen de modifier avec une fonction des variables extérieures à celle-ci. On utilise pour cela des variables globales avec le mot-clé **global**.

Le code suivant permet de modifier la variable x extérieure à la fonction :

```
x=3
def f():
    global x
    x+=2
    print(x)

f() # affiche 5
print(x) # affiche 5
```

Ce procédé est à éviter car il nuit à la portabilité d'une fonction.