# 1 Présentation générale

Un langage de programmation est nécessaire pour l'écriture des programmes : un langage simple d'usage, interprété, concis, libre et gratuit, multiplateforme, largement répandu, riche de bibliothèques adaptées aux thématiques étudiées et bénéficiant d'une vaste communauté d'auteurs dans le monde éducatif est nécessaire. Au moment de la conception de ce programme, le langage choisi est Python version 3 (ou supérieure).


Au collège (cycle 4), les élèves ont découvert et pratiqué les éléments fondamentaux d'algorithmique et de programmation. Le programme de seconde de mathématiques approfondit l'apprentissage de la programmation. Une coordination avec le cours de mathématiques est donc nécessaire pour déterminer à quel moment des éléments de programmation peuvent être utilisés en sciences numériques et technologie.

# 2 Capacités attendues

| **Contenus** | **Capacités attendues** |
|---|---|
| Affectations, variables<br>Séquences<br>Instructions conditionnelles<br>Boucles bornées et non bornées<br>Définitions et appels de fonctions | Écrire et développer des programmes pour répondre à des problèmes et modéliser des phénomènes physiques, économiques et sociaux. |

# 3 Quelques bases de programmation en Python

## 3.1 Variables et affectations

**Définition 3.1 (Variable)** En informatique, une *variable* est une donnée que l'ordinateur stocke dans un espace mémoire.

**Définition 3.2 (Affectation)** *L'affectation* est le fait de donner une « valeur » à une variable.

**Exemple 3.3** *A prend la valeur 7* s'écrit en algorithmique $A \leftarrow 7$ et se traduit en Python par `A = 7`.

Pour affecter une valeur à une variable, on utilise le symbole =.

- Le nom d'une variable est une succession de caractères alphanumériques ne contenant aucun espace, aucun caractère spécial sauf l'underscore _ et ne commençant jamais par un nombre. Il peut contenir des majuscules ou des minuscules.
- La variable affectée prend le type de la valeur que l'on affecte. Par exemple, si l'on affecte `3` à `x`, la variable `x` est de type `int`.

On distingue deux types d'affectations : les affectations simples et les affectations multiples.

### 3.1.1 Les affectations simples

Pour pouvoir affecter une valeur à une variable, on utilise la syntaxe suivante :

```
nom_variable = valeur
```

**Exemple 3.4** Les instructions

```
a = 3; a += 5; a
a = [1,2]; a += ["A"]; a
a = "abc"; a += "d"; a
a = "A"; a *= 3; a
a = 5; a -= 1; a
a = 15; a /= 2; a
a = 5; a //= 3; a
a = 18; a %= 7; a
```

renvoient respectivement `8`, `[1,2,"A"]`, `"abcd"`, `"AAA"`, `4`, `7.5`, `1` et `4`.

### 3.1.2 Les affectations multiples

On peut également procéder à une affectation simultanée de plusieurs variables. Pour se faire, on utilise la syntaxe suivante :

```
nom_variable_1,nom_variable_2,...,nom_variable_n = liste de n valeurs
```

Les $n$ valeurs sont affectées de gauche à droite aux variables déclarées à gauche de l'égalité. La liste de ces $n$ valeurs peut être écrite sous l'une des formes suivantes :

- directement et séparées par des virgules : `val_1,val_2,...,val_n`,
- avec un tuple : `(val_1,val_2,...,val_n)`,
- avec une liste : `[val_1,val_2,...,val_n]`.

**Exemple 3.5** Considérons les instructions suivantes :

```
a,b = 1,"essai"; a; b
a,b,c = (1,3,-5); a; b; c
a,b = ["essai",True]; a; b
```

La première ligne renvoie `1` pour `a` et `"essai"` pour `b`.
La deuxième ligne renvoie `1` pour `a`, `3` pour `b` et `-5` pour `c`.
La troisième ligne renvoie `"essai"` pour `a` et `True` pour `b`.

L'affectation simultanée permet également d'échanger des valeurs facilement. Ainsi, la syntaxe

```
a,b = b,a
```

permet d'échanger les valeurs des variables `a` et `b`.

### 3.1.3 Les types de données

Dans la machine l'information étant binaire, le type des variables permet de la traduire en utilisant le bon « décodeur ». En Python il n'y a pas de déclaration nécessaire : Python attribue le type au moment de l'exécution de l'instruction d'affectation.

**Exemple 3.6**

- `A = 17` : A est du type `int` c'est-à-dire un nombre entier ;
- `A = 17.0` : A est du type `float` c'est-à-dire un nombre « à virgule » ;
- `A = 'bonjour'` : A est du type `str` c'est-à-direune chaîne de caractères ;
- `A = True` : A est du type `bool` c'est-à-dire une donnée booléenne (soit vrai, soit faux) ;
- `A = [1 , 2.5, 7, 'a']` : A est du type `list` c'est-à-dire une liste

Les types que nous utiliserons le plus souvent sont les suivants :

| nombre entier | nombre flottant | vide | booléen | range | liste | tuple |
|---|---|---|---|---|---|---|
| `int` | `float` | `NoneType` | `bool` | `range` | `list` | `tuple` |

Pour obtenir le type d'une donnée ou d'une variable, on utilise la fonction `type()`.

**Exemple 3.7** L'instruction `type(5)` renvoie `<class 'int'>`, l'instruction `type(a)` quand la variable `a` contient la valeur `4.5` renvoie `<class 'float'>`.

**Le transtypage** Pour pouvoir modifier certains types de données ou pouvoir appliquer certaines fonctions ou méthodes, il est parfois nécessaire de passer d'un type à un autre. On procède alors à un *transtypage*. Pour ce faire, il suffit d'appliquer à la donnée que l'on souhaite transtyper la fonction donnant le nouveau type.

**Exemple 3.8**

- Si `x` est un flottant, l'instruction `int(x)` transtype `x` en un entier (en le tronquant à l'unité). Par exemple, `int(7.2)` renvoie `7`.
- Si `n` est un entier, l'instruction `float(n)` transtype `n` en un flottant. Par exemple, `float(5)` renvoie `5.0`.

Certains transtypages sont toujours valides. Par exemple, l'instruction `str(x)` renvoie toujours une chaîne de caractères, quelque soit le type de la donnée `x`. En revanche, certains transtypages peuvent ne pas fonctionner suivant le type de données utilisées.

**Exemple 3.9** L'instruction `int("3")` renvoie `3` et donc permet de transtyper la chaîne de caractères `"3"` en un entier. En revanche, l'instruction `int("3.4")` renvoie une erreur.

### 3.1.4 Opérations numériques usuelles

- **Opérations algébriques.**

| addition | soustraction | multiplication | division | puissance n$^{\text{ème}}$ de x |
|---|---|---|---|---|
| `+` | `-` | `*` | `/` | `x**n` |

**Remarque 3.10** Attention, les opérations doivent toujours être écrites. Par exemple, on écrit `2*x` et non pas `2x`.

- **Opérations avec les nombres entiers.** Soient `a` et `b` deux nombres entiers.

| quotient de `a` par `b` | reste de `a` par `b` |
|---|---|
| `a//b` | `a%b` |

**Exemple 3.11** L'instruction `5//2` renvoie `2` et l'instruction `5%2` renvoie `1`.

- **Valeur absolue et arrondis.**

| valeur absolue de x | arrondi de x avec n décimales |
|---|---|
| `abs(x)` | `round(x,n)` |

## 3.2 Séquences d'instructions

**Définition 3.12** Une *séquence d'instructions* est une suite finie d'instructions algorithmiques.

## 3.3 Instructions conditionnelles

### 3.3.1 Les tests

**Définition 3.13** Les données de type `bool` sont des données qui ne peuvent prendre que deux valeurs : `True` ou `False`.

Ce type de données apparaît lorsque l'on teste si une condition est réalisée. Par exemple, l'instruction `2 < 8` renvoie la valeur `True`, alors que l'instruction `5 < 1` renvoie la valeur `False`. On distingue deux sortes de tests : les tests avec les opérateurs de comparaison et les tests avec les opérateurs logiques.

⋆ **Les opérateurs de comparaison**

| égal | différent | inférieur | supérieur | inférieur strict | supérieur strict |
|---|---|---|---|---|---|
| == | != | <= | >= | < | > |

**Exemple 3.14** Les instructions

```
3 == 5; "ab" != "cd"; 5 <= 8; 4 >= -3; 3 < 1; "e" > "a"
```

renvoient respectivement `False`, `True`, `True`, `True`, `False` et `True`.

**Remarque 3.15** Les opérateurs == et != peuvent aussi s'appliquer à des listes, des tuples, etc...

**Remarque 3.16** Les opérateurs <=, >=, < et > s'appliquent bien sûr aux entiers et flottants, mais également aux chaînes de caractères. Dans ce cas, l'ordre utilisé est celui de l'ordre alphabétique.

**Remarque 3.17** Aux opérateurs de comparaison précédents s'ajoute l'opérateur `in` que nous avons déjà rencontré au paragraphe 3 et qui permet de tester si un élément appartient à une structure.

⋆ **Les opérateurs logiques**

| ou | et | non |
|---|---|---|
| or | and | not |

**Exemple 3.18** Les instructions

```
3==3 or 9>24; 3==3 and 9>24; not(3==3)
```

renvoient respectivement `True`, `False` et `False`.

**Remarque 3.19** On peut bien sûr combiner les opérateurs logiques entre eux, mais attention aux ordres de priorité...

**Exemple 3.20** Les instructions

```
(3 < 4 or "a" < "bc") and -2 >1 ; 3 < 4 or ("a" < "bc" and -2 >1)
```

renvoient respectivement `False` et `True`.

### 3.3.2 La structure `si ... alors ...`

La structure algorithmique

```
si une condition est vraie alors
   séquence d'instructions
fin si
```

s'écrit

```
if condition:                  # ne pas oublier les :
   séquence d'instructions     # indentation obligatoire
# pour terminer, on stoppe l'indentation
```

**Exemple 3.21** Le code Python suivant permet de renvoyer la chaîne de caractères `Le nombre est positif` si le nombre `x` est positif :

```
if x >= 0:
  return "Le nombre est positif"
```

### 3.3.3 La structure `si ... alors ... sinon ...`

La structure algorithmique

```
si une condition est vraie alors
   séquence d'instructions 1
sinon
   séquence d'instructions 2
fin si
```

s'écrit

```
if condition:                  # ne pas oublier les :
   séquence d'instructions 1   # indentation obligatoire
else:                          # ne pas oublier les :
   séquence d'instructions 2   # indentation obligatoire
# pour terminer, on stoppe l'indentation
```

**Exemple 3.22** Le code Python suivant permet de renvoyer la chaîne de caractères `"Le nombre est négatif"` ou `"Le nombre est positif"` suivant le signe de `x` :

```
if x<=0:
  return "Le nombre est négatif"
else:
  return "Le nombre est positif"
```

## 3.4 Boucles bornées et non bornées

### 3.4.1 La boucle *Pour* ou *boucle bornée* ou *boucle itérative*

La structure

```
pour i allant de 0 à n-1 faire
   séquence d'instructions
fin pour
```

s'écrit

```
for i in range(n):          # ne pas oublier les :
   séquence d'instructions  # indentation obligatoire
# pour terminer, on stoppe l'indentation
```

**Définition 3.23** Les données de type `range` permettent d'énumérer une liste ordonnée de nombres entiers. Pour définir une donnée de ce type, on utilise la fonction `range()` qui admet de un à trois paramètres :

- `range(n)` : cette fonction renvoie une structure constituée de tous les nombres entiers compris entre `0` et `n - 1` ; le paramètre `n` est un nombre entier. Si `n` $\leqslant$ `0`, la structure est vide.
- `range(m,n)` : cette fonction renvoie une structure constituée de tous les nombres entiers compris entre `m` et `n - 1` ; les paramètres `m` et `n` sont des nombres entiers. Si `n` $\leqslant$ `m`, la structure est vide.
- `range(m,n,p)` : cette fonction renvoie une structure constituée de tous les nombres entiers compris entre `m` et `|n| - 1` avec un pas de `p` ; les paramètres `m`, `n` et `p` sont des nombres entiers et `p` est non nul.
  - Si `n` $\leqslant$ `m` et `p` $>$ `0` ou si `m` $\leqslant$ `n` et `p` $<$ `0`, la structure est vide.
  - Si `m` $<$ `n` et `p` $>$ `0`, la structure est constituée de nombres entiers croissants.
  - Si `n` $<$ `m` et `p` $<$ `0`, la structure est constituée de nombres entiers décroissants.

**Exemple 3.24** Une somme de 2000 € est placée à 2%. Le code Python suivant calcule la somme disponible au bout de 20 ans :

```
S = 2000                # on initialise la somme S
for i in range(20):
  S = S * 1.02          # on calcule l'évolution des sommes S au cours des 20 ans
```

### 3.4.2 La boucle *Tant que* ou *boucle non bornée* ou *boucle conditionnelle*

La structure

```
tant que une condition reste vraie faire
   séquence d'instructions
fin tant que
```

s'écrit

```
while condition:          # ne pas oublier les :
   séquence d'instructions# indentation obligatoire
# pour terminer, on stoppe l'indentation
```

**Exemple 3.25** Une somme de $2000$ € est placée à $2\%$. Le code Python suivant calcule au bout de combien d'années on dispose d'une somme d'au moins $5000$ € (le résultat est stocké dans la variable n) :

```
S = 2000              # on initialise la somme S
n = 0                 # on initialise le compteur « manuel » n
while S < 5000:
   S = S * 1.02       # on calcule la nouvelle somme disponible chaque année
   n = n + 1          # on incrémente le compteur de 1
```

## 3.5 Définition et appels de fonctions

En informatique, une fonction est une suite d'instructions (sorte de sous-programme) que l'on peut appeler au besoin. On définit une fonction pour éviter les répétitions, ou pour décomposer un programme complexe en sous-tâches plus simples. Les fonctions permettent de rendre un programme plus lisible.

### 3.5.1 Création d'une fonction

La syntaxe pour la création d'une fonction est la suivante :

```
def nom_fonction(liste_paramètres):       # ne pas oublier les :
   """ documentation de la fonction """   # chaîne de caractères entre triple guillemets
   séquence d'instructions                # indentation obligatoire
   return résultat                        # renvoie le résultat de la fonction
# pour terminer, on stoppe l'indentation
```

**Remarque 3.26** La liste des paramètres peut être vide.

**Remarque 3.27** La documentation de la fonction est très importante. Elle fournit des informations sur les types des paramètres, sur ce qu'elle fait et sur ce qu'elle renvoie. On accède à cette documentation en utilisant l'instruction help(*nom_fonction*).

**Remarque 3.28** L'instruction return stoppe l'exécution de la fonction et renvoie une ou plusieurs valeurs. De cette façon on peut récupérer le résultat renvoyé par une fonction. Si plusieurs valeurs sont renvoyées, elles le sont sous forme d'un tuple.

**Remarque 3.29** Si l'instruction return est omise, la fonction renvoie None et on parle alors de *procédure*.

**Exemple 3.30** La fonction suivante calcule le maximum de deux valeurs :

```
def maximum(a,b):
   """
   a et b sont deux nombres flottants
   Renvoie le maximum de a et b
   """
   if a < b:
      return b
   return a
```

### 3.5.2 Appel d'une fonction

Pour utiliser une fonction, il faut l'appeler et lui passer des valeurs par arguments si elle possède des paramètres. Les valeurs passées doivent bien sûr être dans le même ordre que les paramètres et du même type que ceux-ci.

**Exemple 3.31** Pour déterminer avec la fonction de l'exemple 6.5 le maximum de 5 et 28, on tape l'instruction `maximum(5,28)` et celle-ci renvoie `28`.

Les paramètres d'une fonction peuvent être eux-mêmes des fonctions.
Il existe un grand nombre de fonction déjà définies dans le langage `Python` que l'on peut utiliser. Il est également possible de charger des *bibliothèques* ou *modules* contenant des fonctions.

### 3.5.3 Importation d'une bibliothèque

Pour pouvoir utiliser une bibliothèque, il est nécessaire de l'importer. Pour se faire, on peut utiliser l'une des syntaxes suivantes :

```
import nom_bibliothèque
import nom_bibliothèque as alias
from nom_bibliothèque import liste de fonctions
from nom_bibliothèque import *
```

- La première méthode d'importation indique à `Python` quelle bibliothèque on veut utiliser. Pour utiliser une fonction qui y est intégrée, on utilise la syntaxe

  ```
  nom_bibliothèque.nom_fonction
  ```

  **Exemple 3.32** Pour calculer $\sqrt{2}$ avec la bibliothèque `math`, on tape les instructions

  ```
  import math
  math.sqrt(2)
  ```

  L'un des intérêts de cette méthode d'importation est de bien spécifier à quelle bibliothèque appartient la fonction que l'on utilise.

- La deuxième méthode d'importation est analogue à la première mise à part que l'on donne un alias à la bibliothèque. Pour utiliser une fonction qui y est intégrée, on utilise la syntaxe

  ```
  alias.nom_fonction
  ```

- La troisième méthode d'importation indique à `Python` d'importer directement une liste de fonctions d'une bibliothèque donnée. Pour utiliser une de ces fonctions, on l'appelle directement.

  **Exemple 3.33** Pour calculer $\sqrt{2}$ avec la fonction `sqrt` de la bibliothèque `math`, on tape les instructions

  ```
  from math import sqrt
  sqrt(2)
  ```

  L'un des intérêts de cette méthode d'importation est de n'importer que les fonctions nécessaires.

- La quatrième méthode d'importation est analogue à la précédente, mise à part que l'on demande à `Python` d'importer une fois pour toute tout le contenu d'une bibliothèque. On peut alors utiliser directement toutes les fonctions qui y sont intégrées en les appelant.

Le tableau ci-dessous regroupe les principales bibliothèques utiles en SNT :

| nom de la bibliothèque | description brève |
|---|---|
| `math` | fonctions mathématiques réelles |
| `random` | nombres pseudo-aléatoires |
| `matplotlib.pyplot` | outils mathématiques pour les graphiques |