Chapitre 4 : Langages et programmation

1. Les types de variables

Chaque variable dans Python possède un type :

- **int** : de l'anglais « *integer* », correspond aux entiers relatifs ;
- **float** : de l'anglais « *floatting-point number* » [« nombre à virgule flottante »], correspond à un mode de représentation des nombres réels pour lequel le séparateur entre les parties entières et décimale est un point ;
- **str** : de l'anglais « *string of characters* » [« chaîne de caractères »], correspond à une chaîne de caractères ;
- **bool** : de l'anglais « *boolean value* » [« valeur booléenne »], correspond au résultat d'un test. Une variable de ce type peut prendre deux valeurs : « True » [Vrai] ou « False » [Faux].

Exemples :

```
>>> entier = 5   #la variable « entier » contient un nombre entier de type int
>>> type(entier)
<class 'int'>
>>> reel = 2.5   #la variable « reel » contient un nombre à virgule flottante de type float
>>> type(reel)
<class 'float'>
>>> chaine = "Spécialité NSI"
>>> type(chaine)
<class 'str'>
```

2. Les opérateurs booléens

Les **opérateurs booléens** sont des opérateurs de *comparaison* qui retournent un objet de type booléen : « `True` » ou « `False` ».

Soit x et y des variables de type `int` ou `float`, les opérateurs de comparaison sont :

Opérateur	Rôle
`x < y`	x est-il inférieur à y ?
`x <= y`	x est-il inférieur ou égal à y ?
`x > y`	x est-il supérieur à y ?
`x >= y`	x est-il supérieur ou égal à y ?
`x == y`	x est-il égal à y ?
`x != y`	x est-il différent y ?

Exemples :

```
>>> x = 3
>>> y = 5
>>> print(x < y)
True
>>>
>>> print(x <= y)
True
>>> x > y
False
```

ATTENTION : pour tester une égalité on utilise le double signe égal « == ». Tout simplement car le signe égal seul « = » est déjà utilisé pour l'affectation de variables.

3. Les fonctions

3.1. Définition

Une fonction permet d'écrire des instructions qui vont pouvoir être réutilisées, cela évitera de réécrire constamment les mêmes instructions et de **rendre le programme plus lisible**. Elle peut recevoir un ou plusieurs **arguments** (valeurs ou variables) et peut renvoyer une ou plusieurs valeurs de retour.
Une fonction possède un nom pour pouvoir être appelée et il est possible de lui communiquer des arguments :

```
def nom_de_la_fonction(arguments séparés par des virgules):
    instruction 1
    instruction 2
    ...
    return objet_renvoyé_par_la_fonction
```

Pour appeler la fonction, il suffit de taper son nom avec les arguments entre parenthèse :

```
nom_de_ma_fonction(arg1, arg2,...)
```

Exemple :

La vitesse d'un objet est définie par le rapport de la distance parcourue par la durée mise pour la parcourir :

$$v = \frac{d}{\Delta t}$$

Nous allons créer une fonction « `vitesse()` » qui calcule sa valeur à partir des paramètres distance et durée :

```
#définition de la fonction
def vitesse(distance, duree):
    return distance / duree
```

Appel et sortie :

```
>>> #on appelle la fonction avec une distance de 50 m et une durée de 10s
>>> vitesse(50, 10)
5.0
```

1.1. Prototyper une fonction

Pour expliquer le fonctionnement d'une fonction, on lui ajoute un **prototype** juste sous la ligne de définition. En Python les prototypes sont appelés *docstrings*. On peut y accéder dans le code source ou simplement en utilisant la fonction `help()`.

→ Le prototype doit décrire le **rôle de la fonction**, le **type des paramètres en entrée** et **le type de la valeur de retour**.

Procédure :

❶ Définir l'objectif / le rôle de la fonction (ou du programme)
❷ Identifier les paramètres (entrées, sorties, type, rôle)
❸ Identifier les préconditions (conditions d'utilisation de la fonction ou du programme)
❹ Identifier les postconditions (effet de la fonction ou du programme)

→ On écrit ensuite le tout en en-tête de la fonction ou du programme, entre """ et """.

Exemple :

```
#définition de la fonction
def minimum(liste):
    """
       Renvoie le minimum d'une liste d'entiers

       Paramètres :
       ------------

       liste : liste d'entiers en argument
       min : un entier

       Sortie :
       --------
       Entier : le minimum de la liste
    """
    min = liste[0]
    for i in liste:
        if i < min:
            min = i
    return min
```

Prototypage

→ En tapant `help(minimum)`, on obtient une **documentation** sur la fonction : les informations situées entre les """ … """ dans le code du programme.

1.2. Renvoi de valeur

Tout ce qui se passe à l'intérieur d'une fonction reste à l'intérieur de la fonction, mais par contre les `print()` sont des fonctions globales qui agissent à l'extérieur de la fonction.

En utilisant l'instruction « `return` », on pourra récupérer le résultat des instructions traitées par une fonction lorsqu'on l'appellera, et qu'on assignera éventuellement des valeurs de retour à des variables (globales ou non).

Exemple n°1 :

```
1  def vitesse(distance, duree):
2      return distance / duree
```

Appel et sortie :

```
1  >>> #on appelle la fonction avec une distance de 50 m et une durée de 10s
2  >>> vitesse(50, 10)
3  5.0 #la fonction renvoie le résultat du calcul de la vitesse
```

Exemple n°2 :

```
1  def compare(distance1, distance2):
2      if distance1 < distance2:
3          return True
4      else:
5          return False
```

Appel et sortie :

```
1  >>> #on appelle la fonction avec une distance de 50 m et une durée de 10s
2  >>> compare(50, 10)
3  False #la fonction renvoie le booléen « True »
```

4. Mise au point d'un programme

En informatique, un **paradigme** est une façon de programmer, un modèle qui oriente notre manière de penser pour formuler et résoudre un problème.

Certains langages sont conçus pour supporter un paradigme en particulier (Smalltalk et Java supportent la programmation orientée objet (paragramme de terminale), tandis que Haskell est conçu pour la programmation fonctionnelle), alors que d'autres supportent des paradigmes multiples (à l'image de C++, Common Lisp, OCaml, Python, Ruby ou Scheme).

La conception d'un programme doit suivre un certain nombre d'étapes :

❶ Comprendre le problème (faire des essais à la main ...)
❷ Mettre au point un algorithme et s'interroger sur sa :

- Terminaison (être certain que l'algorithme se termine)
- Correction (être certain que le résultat est la (ou une) solution du problème)
- Complétude (être certain que l'algorithme reste correct si on modifie par exemple le nombre d'entrées)
- Complexité (nombres d'opérations que fait l'algorithme)

❸ Écrire le programme (dans un langage adapté au problème)

- Choisir les bibliothèques nécessaires
- Choisir des noms de variables et leur portée (globale ou locale)
- Spécifier des sous programmes (éventuellement)
- Placer des tests (pour voir si les parties du programme testés font bien ce que l'on espère...)
- Essayer d'anticiper les « bugs » possibles (division par 0 , portée des variables...)
- Modifier l'algorithme si besoin, pour l'optimiser, ...

❹ Faire de nombreux tests...

3.1. Les tests

Pour s'assurer qu'une fonction donne les bons résultats et reçoive en argument(s) le bon type de données, on utilise des tests d'**assertion** *(affirmation en anglais)*, avec l'instruction « `assert` », qui renvoient une `AssertionError` en cas d'erreur.

Exemple :

```
1  def test_pH(pH):
2      """ teste le cartactère acide, basique ou neutre d'une solution
3          Effectue un affichage mais ne renvoie rien
4          Teste si le pH est un entier ou flottant et est compris entre 0 et 14
5      """
6      assert type(pH) == int or type(pH) == float
7      assert pH >= 0 and pH <= 14
8
9      if pH < 7:
10         print("Solution acide")
11     elif pH == 7:
12         print("Solution neutre")
13     else:
14         print("Solution basique")
```

La fonction ci-dessus contient deux tests d'assertion :

```
6      assert type(pH) == int or type(pH) == float
7      assert pH >= 0 and pH <= 14
```

- Le premier test (ligne 6) vérifie que le paramètre « pH » reçu en argument est bien un nombre entier (« `int` ») ou réel (« `float` ») ;
- Le second test (ligne 7) vérifie que la valeur passée en paramètre (« `pH` ») à la fonction respecte bien les limites d'utilisation de la fonction.

2.1. Les bibliothèques

Les bibliothèques ou `modules` permettent de **rassembler et d'organiser les diverses fonctionnalités** d'un programme afin de pouvoir les importer ultérieurement.

Elles constituent un outil important dans la mise au point de programme complexe en permettant de séparer le code en diverses parties, chacune de ces parties ayant un rôle bien spécifique.

De nombreuses bibliothèques sont fournies avec Python, comme par exemple les modules `math` ou `random`. On les appelle les bibliothèques standards.

On peut également en utiliser d'autres non fournies avec Python, mais facilement installables comme `pandas` pour le traitement de données, ou `matplotlib` pour le tracé de graphiques.

Exemple n°1 :

```
1  >>> import math
2  >>> sqrt(4)
3  2.0 #la fonction « sqrt » renvoie la racine carrée de 4
```

Exemple n°2 :

```
1  import random
2  random.random()
```

Sortie :

```
1  0.560650748358627 #génération d'un nombre aléatoire
```

On peut accéder à la documentation d'un module directement à partir de Python avec la fonction `help`, mais on préférera tout de même la documentation en ligne lorsque cela est possible :

- https://docs.python.org/fr/3/ (Documentation Python en ligne)

Programme :

Contenus	Capacités attendues	Commentaires
Constructions élémentaires	Mettre en évidence un corpus de constructions élémentaires.	Séquences, affectation, conditionnelles, boucles bornées, boucles non bornées, appels de fonction.
Diversité et unité des langages de programmation	Repérer, dans un nouveau langage de programmation, les traits communs et les traits particuliers à ce langage.	Les manières dont un même programme simple s'écrit dans différents langages sont comparées.
Spécification	Prototyper une fonction. Décrire les préconditions sur les arguments. Décrire des postconditions sur les résultats.	Des assertions peuvent être utilisées pour garantir des préconditions ou des postconditions.
Mise au point de programmes	Utiliser des jeux de tests.	L'importance de la qualité et du nombre des tests est mise en évidence. Le succès d'un jeu de tests ne garantit pas la correction d'un programme.
Utilisation de bibliothèques	Utiliser la documentation d'une bibliothèque.	Aucune connaissance exhaustive d'une bibliothèque particulière n'est exigible.

Sitographie :

- https://isn-icn-ljm.pagesperso-orange.fr/NSI/co/Langages_2.html
- https://www.lyceum.fr/1g/nsi/7-langages-et-programmation
- https://www.lyceum.fr/1g/nsi/7-langages-et-programmation/8-mise-au-point-dun-programme