Polycopié pour le cours de Python

Romain Tavenard

Chapitre 1

Introduction

Ce document est une tentative de polycopié associé au module de Python pour la deuxième année de licence MIASHS de l'Université de Rennes 2. Il est distribué librement et se veut évolutif, n'hésitez donc pas à faire vos remarques à son auteur dont vous trouverez le contact sur sa page web.

Durant la lecture de ce polycopié, vous trouverez des blocs de code tels que celui-ci :

```
def f(v):
    return v ** 2

x = 5
y = f(3 * x + 2)
print(y)
# [Sortie] 77
```

Nous prendrons notamment l'habitude de reporter les valeurs affichées par l'exécution du programme considéré dans un terminal avec la syntaxe utilisée à la dernière ligne du code ci-dessus.

TODO : quelques mots sur langage interprété, console Python vs script ? dire aussi que les exemples considérés ici se basent sur Python3 ? (notamment pour les histoires de division d'entiers)

Chapitre 2

Structures de données et structures de contrôle

Dans ce chapitre, nous allons nous intéresser aux éléments de base de la syntaxe Python : les structures de données d'une part et les structures de contrôle d'autre part. Les structures de données vont nous permettre de stocker dans la mémoire de l'ordinateur (dans le but de les traiter ensuite) des données tandis que les structures de contrôle vont servir à définir les interactions que nous allons avoir avec ces données.

2.1 Variables

En Python, les structures de données sont toutes des variables, il n'existe pas de constante. Une variable est une association entre un symbole (le nom de la variable) et une valeur, cette dernière pouvant varier au cours de l'exécution du programme. Les variables Python sont typées dynamiquement, ce qui signifie qu'une variable, à un moment donné de l'exécution d'un programme, a un type précis qui lui est attribué, mais que celui-ci peut évoluer au cours de l'exécution du programme.

Les types de base existant en Python sont les suivants :

- `int` : entier ;
- `float` : nombre à virgule ;
- `complex` : nombre complexe (peu utilisé en pratique) ;
- `str` : chaîne de caractères ;
- `bool` : booléen (pouvant prendre les valeurs `True` ou `False`).

En Python, le type d'une variable n'est pas déclaré par l'utilisateur : il est défini par l'usage (la valeur effective que l'on décide de stocker dans la variable en

question).

Par exemple, l'instruction suivante en Python attribue la valeur `12` à la variable `v`, qui devient donc automatiquement de type entier :

```
v = 12
```

Ainsi, les instructions suivantes ont toutes une incidence sur le type des variables considérées :

```
v = 12     # v est alors de type entier
c = "abc"  # c est de type chaîne de caractères
d = 'abc'  # d est également de type chaîne de caractères
           # les contenus de c et d sont identiques
v = 12.    # v change de type et est désormais de type nombre à virgule
```

Pour vérifier le type d'une variable, il suffit d'utiliser la fonction `type(.)` de la librairie standard :

```
print(type(v))  # la fonction print(.) permet d'afficher
                # une information dans le terminal
# [Sortie] <class 'float'>
```

Comme le montrent les exemples précédents, pour pouvoir utiliser des variables, on doit leur donner un nom (placé à gauche du signe égal dans l'opération d'affectation). Ces noms de variables doivent respecter certaines contraintes :

- ils doivent débuter par une lettre (minuscule ou majuscule, peu importe) ou par le symbole `_` ;
- ils ne doivent contenir que des lettres, des chiffres et des symboles `_` ;
- ils ne doivent pas correspondre à un quelconque mot réservé du langage Python, dont voici la liste :

  ```
  and del for is raise assert elif from lambda return break else global
  not try nonlocal True False class except if or while continue import
  pass yield None def finally in
  ```

- ils ne doivent pas correspondre à des noms de fonction de la librairie standard de Python (cette dernière condition n'est en fait qu'une bonne pratique à observer) : vous apprendrez au fur et à mesure les noms de ces fonctions.

Les noms de variable en Python sont sensibles à la casse, ainsi les variables `maVariable` et `mavariable` ne pointent pas sur les mêmes données en mémoire. Pour s'en convaincre, on peut exécuter le code suivant :

```
mavariable = 12
maVariable = 15
print(mavariable)
# [Sortie] 12
print(maVariable)
# [Sortie] 15
```

Comme on l'a vu plus haut, on utilise en Python l'opérateur `=` pour assigner une valeur à une variable. La sémantique de cet opérateur est la suivante : "assigner la valeur contenue dans le membre de droite à la variable du membre de gauche". Ainsi, il est tout à fait valide d'écrire, en Python :

```
x = 3.9 * x * (1 - x)
```

Pour exécuter cette instruction, l'interpréteur Python commencera par évaluer le membre de droite en utilisant la valeur courante de la variable `x`, puis affectera la valeur correspondant au résultat de l'opération `3.9 * x * (1 - x)` dans la variable `x`.

On le voit dans l'exemple précédent, pour manipuler des variables, on utilisera des opérateurs (dont les plus connus sont les opérateurs arithmétiques). Le tableau suivant dresse une liste des opérateurs définis pour les variables dont le type est l'un des types numériques (entier, nombre à virgule, nombre complexe) :

Opérateur	Opération
`+`	Addition
`-`	Soustraction
`*`	Multiplication
`/`	Division
`**`	Élévation à la puissance
`%`	Modulo (non défini pour les nombres complexes)

De plus, pour chacun de ces opérateurs, il existe un opérateur associé qui réalise successivement l'opération demandée puis l'affectation de la nouvelle valeur à la variable en question. Ainsi, l'instruction suivante :

```
x = x + 2
```

qui ajoute 2 à la valeur courante de `x` puis stocke le résultat du calcul dans `x` peut se réécrire :

```
x += 2
```

Ceci est purement un raccourci de notation, s'il ne vous semble pas évident à maîtriser au premier abord, vous pouvez vous en passer et toujours utiliser la notation `x = x + 2`.

Enfin, lorsque l'évaluation d'une expression implique plusieurs opérateurs, les règles de priorité sont les suivantes (de la priorité maximale à la priorité minimale) :

1. parenthèses ;
2. élévation à la puissance ;
3. multiplication / division ;
4. addition / soustraction ;

5. de gauche à droite.

Pour prendre un exemple concret, pour évaluer l'expression :

```
3.9 * x * (1 - x)
```

on commencera par évaluer le contenu de la parenthèse puis, les 2 opérations restantes étant toutes des multiplications, on les effectuera de gauche à droite.

De plus, lorsqu'une opération est effectuée entre deux variables de types différents, le type le plus générique est retenu. Par exemple, si l'on multiplie un entier par un nombre à virgule, le résultat sera de type `float`. De même, le résultat de l'addition entre un nombre complexe et un nombre à virgule est un complexe.

Attention. Comme indiqué en introduction, ce polycopié suppose que vous utilisez Python dans sa version 3. Il est à noter qu'il existe une différence importante entre Python 2 et Python 3 dans la façon d'effectuer des opérations mêlant nombres entiers et flottants. Par exemple, l'opération suivante :

```
x = 2 / 3
```

stockera, en Python 2, la valeur 0 (résultat de la division *entière* de 2 par 3) dans la variable `x` alors qu'en Python 3, la division flottante sera effectuée et ainsi `x` contiendra `0.666666...` En Python 3, si l'on souhaite effectuer une division entière, on pourra utiliser l'opérateur `//` :

```
print(2 // 3)
# [Sortie] 0
```

2.2 Structures de contrôle

Un programme est une séquence d'instructions dont l'ordre doit être respecté. Au-delà de cet aspect séquentiel, on peut souhaiter :

- n'effectuer certaines instructions que si une condition est vérifiée (structures conditionnelles) ;
- répéter certaines instructions (boucles) ;
- factoriser une sous-séquence d'instructions au sein d'une fonction pour pouvoir y faire appel à plusieurs reprises dans le programme (fonctions).

2.2.1 Structures conditionnelles

On peut donc indiquer à un programme de n'exécuter une instruction (ou une séquence d'instructions) que si une certaine condition est remplie, à l'aide du mot-clé `if` :

```
x = 12
if x > 0:
```

```
    print("X est positif")
    x = 4
y = 2
# [Sortie] X est positif
```

On remarque ici que la condition est terminée par le symbole `:`, de plus, la séquence d'instructions à exécuter si la condition est remplie est *indentée*, cela signifie qu'elle est décalée d'un "cran" (généralement une tabulation ou 4 espaces) vers la droite. Cette indentation est une bonne pratique recommandée quel que soit le langage que vous utilisez, mais en Python, c'est même une obligation (sinon, l'interpréteur Python ne comprendra pas où commence et où se termine la séquence à exécuter sous condition).

Dans certains cas, on souhaite exécuter une série d'instructions si la condition est vérifiée et une autre si elle ne l'est pas. Pour cela, on utilise le mot-clé `else` comme suit :

```
x = -1
if x > 0:
    print("X est positif")
    x = 4
else:
    print("X est négatif")
y = 5
# [Sortie] X est négatif
```

Là encore, on remarque que l'indentation est de rigueur pour chacun des deux blocs d'instructions. On note également que le mot-clé `else` se trouve au même niveau que le `if` auquel il se réfère.

Enfin, de manière plus générale, il est possible de définir plusieurs comportements en fonction de plusieurs tests successifs, à l'aide du mot-clé `elif` :

```
x = -1
if x > 0:
    print("X est positif")
    x = 4
elif x > -2:
    print("X est compris entre -2 et 0")
elif x > -4:
    print("X est compris entre -4 et -2")
else:
    print("X est inférieur à -4")
y = 5
# [Sortie] X est compris entre -2 et 0
```

Pour utiliser ces structures conditionnelles, il est important de maîtriser les différents opérateurs de comparaison mis à vos disposition en Python, dont voici une liste non exhaustive :

Opérateur	Comparaison effectuée	Exemple
`<`	Plus petit que	`x < 0`
`>`	Plus grand que	`x > 0`
`<=`	Plus petit ou égal à	`x <= 0`
`>=`	Plus grand ou égal à	`x >= 0`
`==`	Égal à	`x == 0`
`!=`	Différent de	`x != 0`
`is`	Test d'égalité pour le cas de la valeur `None`	`x is None`
`is not`	Test d'inégalité pour le cas de la valeur `None`	`x is not None`
`in`	Test de présence d'une valeur dans une liste	`x in [1, 5, 7]`

2.2.2 Boucles

Il existe, en Python comme dans une grande majorité des langages de programmation, deux types de boucles :

— les boucles qui s'exécutent tant qu'une condition est vraie ;
— les boucles qui répètent la même série d'instructions pour différentes valeurs d'une variable (appelée *variable de boucle*).

Les premières ont une syntaxe très similaire à celle des structures conditionnelles simples :

```
x = 0
while x <= 10:
    print(x)
    x = 2 * x + 2
y = 2
# [Sortie] 0
# [Sortie] 2
# [Sortie] 6
```

On voit bien ici, en analysant la sortie produite par ces quelques lignes, que le contenu de la boucle est répété plusieurs fois. En pratique, il est répété jusqu'à ce que la variable `x` prenne une valeur supérieure à 10 (14 dans notre cas). Il faut être très prudent avec ces boucles `while` car il est tout à fait possible de créer une boucle dont le programme ne sortira jamais, comme dans l'exemple suivant :

```
x = 2
y = 0
while x > 0:
    y = y - 1
y = 2
```

En effet, on a ici une boucle qui s'exécutera tant que `x` est positif, or la valeur de cette variable est initialisée à 2 et n'est pas modifiée au sein de la boucle, la

condition sera donc toujours vérifiée et le programme ne sortira jamais de la boucle.

Le second type de boucle repose en Python sur l'utilisation de listes (ou, plus précisément d'itérables) dont nous reparlerons plus en détail dans la suite de cet ouvrage. Sachez pour le moment qu'une liste est un ensemble ordonné d'éléments. On peut alors exécuter une série d'instructions pour toutes les valeurs d'une liste :

```
for x in [1, 5, 7]:
    print(x)
y = 2
# [Sortie] 1
# [Sortie] 5
# [Sortie] 7
```

2.2.3 Fonctions

Nous avons déjà vu dans ce qui précède, sans le dire, des fonctions. Par exemple, lorsque l'on écrit :

```
print(x)
```

on demande l'appel à une fonction, nommée `print` et prenant un **argument** (ici, la variable `x`). La fonction `print` ne retourne pas de valeur, elle ne fait qu'afficher la valeur contenue dans `x` sur le terminal. D'autres fonctions, comme `type` dont nous avons parlé plus haut, **retournent une valeur** et cette valeur peut être utilisée dans la suite du programme, comme dans l'exemple suivant :

```
x = type(1)  # On stocke dans x la valeur retournée par type
y = type(2.)
if x == y:
    z = 1
else:
    z = 2
```

2.2.3.1 Définition d'une fonction

Lorsqu'un ensemble d'instruction est susceptible d'être utilisé à plusieurs occasions dans un ou plusieurs programmes, il est recommandé de l'isoler au sein d'une fonction. Cela présentera les avantages suivants :

— en donnant un nom à la fonction et en listant la liste de ses arguments, on explicite la sémantique de l'ensemble d'instructions en question, ses entrées et sorties éventuelles, ce qui rend le code beaucoup plus lisible ;
— s'il est nécessaire d'adapter à l'avenir le code pour résoudre un *bug* ou le rendre plus générique, vous n'aurez à modifier le code qu'à un endroit

(dans le corps de la fonction) et non pas à chaque fois que le code est répété.

Pour définir une fonction en Python, on utilise le mot-clé `def` :

```
def f(x):
    y = 5 * x + 2
    z = x + y
    return z // 2
```

On a ici défini une fonction

- dont le nom est `f` ;
- qui prend un seul argument, noté `x` ;
- qui retourne une valeur, comme indiqué dans la ligne débutant par le mot-clé `return`.

Il est possible, en Python, d'écrire des fonctions retournant plusieurs valeurs. Pour ce faire, ces valeurs seront séparées par des virgules dans l'instruction `return` :

```
def f(x):
    y = 5 * x + 2
    z = x + y
    return z // 2, y
```

Enfin, en l'absence d'instruction `return`, une fonction retournera la valeur `None`.

Il est également possible d'utiliser le nom des arguments de la fonction lors de l'appel, pour ne pas risquer de se tromper dans l'ordre des arguments. Par exemple, si l'on a la fonction suivante :

```
def affiche_infos_personne(poids, taille):
    print("Poids: ", poids)
    print("Taille: ", taille)
```

Les trois appels suivants sont équivalents :

```
affiche_infos_personne(80, 180)
affiche_infos_personne(taille=180, poids=80)
affiche_infos_personne(poids=80, taille=180)
```

Évidemment, pour que cela soit vraiment utile, il est hautement recommandé d'utiliser des noms d'arguments explicites lors de la définition de vos fonctions.

2.2.3.2 Argument(s) optionnel(s) d'une fonction

Certains arguments d'une fonction peuvent avoir une valeur par défaut, décidée par la personne qui a écrit la fonction. Dans ce cas, si l'utilisateur ne spécifie pas explicitement de valeur pour ces arguments lors de l'appel à la fonction, c'est

la valeur par défaut qui sera utilisée dans la fonction, dans le cas contraire, la valeur spécifiée sera utilisée.

Par exemple, la fonction `print` dispose de plusieurs arguments facultatifs, comme le caractère par lequel terminer l'affichage (par défaut, un retour à la ligne, `"\n"`) :

```
print("La vie est belle")
print("Life is beautiful")
print("La vie est belle", end="--")
print("Life is beautiful", end="--")
# [Sortie] La vie est belle
# [Sortie] Life is beautiful
# [Sortie] La vie est belle--Life is beautiful--
```

Lorsque vous définissez une fonction, la syntaxe à utiliser pour donner une valeur par défaut à un argument est la suivante :

```
def f(x, y=0):  # La valeur par défaut pour y est 0
    return x + 5 * y
```

Attention toutefois, les arguments facultatifs (*ie.* qui disposent d'une valeur par défaut) doivent impérativement se trouver, dans la liste des arguments, après le dernier argument obligatoire. Ainsi, la définition de fonction suivante n'est pas correcte :

```
def f(x, y=0, z):
    return x - 2 * y + z
```

Chapitre 3

Les listes

Chapitre 4

Les chaînes de caractères

Chapitre 5

Les dictionnaires

Chapitre 6

Tester son code

Chapitre 7

Lecture et écriture de fichiers