Introduction à Python III : Contrôle du flux d'instructions

Ahmed Ammar (`ahmed.ammar@fst.utm.tn`)

Institut Préparatoire aux Études Scientifiques et Techniques, Université de Carthage.

Oct 22, 2019

Table des matières

1 Les conditions

1.1 L'instruction `if`

En programmation, nous avons toujours besoin de la notion de condition pour permettre à un programme de s'adapter à différents cas de figure.

Syntaxe

```
if expression: # ne pas oublier le signe de ponctuation ':'
    "bloc d'instructions" # attention à l'indentation (1 Tab ou 4 * Espaces)
# suite du programme
```

— Si l'expression est vraie (`True`) alors le bloc d'instructions est exécuté.
— Si l'expression est fausse (`False`) on passe directement à la suite du programme.

Exemple 1 : Note sur 20. Dans cet exemple nous allons tester si la note entrée par l'utilisateur. Si la note est > 10 on doit recevoir le message : "J'ai la moyenne" sinon il va rien faire.

```
chaine = input("Note sur 20 : ")
note = float(chaine)
if note >= 10.0:
    # ce bloc est exécuté si l'expression (note >= 10.0) est vraie
    print("J'ai la moyenne")

# suite du programme
print("Fin du programme")
```

Note
— Les blocs de code sont délimités par l'indentation.
— L'indentation est obligatoire dans les scripts.

1.2 L'instruction `else`

Une instruction `else` est toujours associée à une instruction `if`.

Syntaxe

```
if expression:
    "bloc d'instructions 1"    # attention à l'indentation (1 Tab ou 4 * Espaces)
else:                          # else est au même niveau que if
    "bloc d'instructions 2"    # attention à l'indentation
# suite du programme
```

— Si l'expression est vraie (`True`) alors le bloc d'instructions 1 est exécuté.
— Si l'expression est fausse (`False`) alors c'est le bloc d'instructions 2 qui est exécuté.

Exemple 2 : moyenne. Dans cet exemple nous allons tester si la note entrée par l'utilisateur. Si la note est > 10 on doit recevoir le message : "J'ai la moyenne" sinon il va afficher "C'est en dessous de la moyenne".

```
chaine = input("Note sur 20 : ")
note = float(chaine)
if note >= 10.0:
    # ce bloc est exécuté si l'expression (note >= 10.0) est vraie
    print("J'ai la moyenne")
else:
    # ce bloc est exécuté si l'expression (note >= 10.0) est fausse
    print("C'est en dessous de la moyenne")
print("Fin du programme")
```

Pour traiter le cas des notes invalides (< 0 ou > 20), on peut imbriquer des instructions conditionnelles :

```
chaine = input("Note sur 20 : ")
note = float(chaine)
if note > 20.0 or note < 0.0:
    # ce bloc est exécuté si l'expression (note > 20.0 or note < 0.0) est vraie
    print("Note invalide !")
else:
    # ce bloc est exécuté si l'expression (note > 20.0 or note < 0.0) est fausse
    if note >= 10.0:
        # ce bloc est exécuté si l'expression (note >= 10.0) est vraie
        print("J'ai la moyenne")
    else:
        # ce bloc est exécuté si l'expression (note >= 10.0) est fausse
        print("C'est en dessous de la moyenne")
print("Fin du programme")
```

Ou bien encore :

```
chaine = input("Note sur 20 : ")
note = float(chaine)
if note > 20.0 or note < 0.0:
    print("Note invalide !")
else:
    if note >= 10.0:
        print("J'ai la moyenne")
        if note == 20.0:
            # ce bloc est exécuté si l'expression (note == 20.0) est vraie
            print("C'est même excellent !")
    else:
        print("C'est en dessous de la moyenne")
        if note == 0.0:
            # ce bloc est exécuté si l'expression (note == 0.0) est vraie
            print("... lamentable !")
print("Fin du programme")
```

1.3 L'instruction `elif`

Une instruction `elif` (contraction de **else if**) est toujours associée à une instruction `if`.

Syntaxe

```
if expression 1:
    "bloc d'instructions 1"
elif expression 2:
    "bloc d'instructions 2"
elif expression 3:
    "bloc d'instructions 3"    # ici deux instructions elif, mais il n'y a pas de limitation
else:
    "bloc d'instructions 4"
# suite du programme
```

— Si l'expression 1 est vraie alors le bloc d'instructions 1 est exécuté, et on passe à la suite du programme.
— Si l'expression 1 est fausse alors on teste l'expression 2 :
— si l'expression 2 est vraie on exécute le bloc d'instructions 2, et on passe à la suite du programme.
— si l'expression 2 est fausse alors on teste l'expression 3, etc.

Le bloc d'instructions 4 est donc exécuté si toutes les expressions sont fausses (c'est le bloc "par défaut").

Parfois il n'y a rien à faire. Dans ce cas, on peut omettre l'instruction `else` :

```
if expression 1:
    "bloc d'instructions 1"
elif expression 2:
    "bloc d'instructions 2"
elif expression 3:
    "bloc d'instructions 3"
# suite du programme
```

L'instruction `elif` évite souvent l'utilisation de conditions imbriquées (et souvent compliquées).

Exemple 3 : moyenne-bis. On peut tester plusieurs possibilités avec une syntaxe beaucoup plus propre avec les instructions `if-elif-else` :

```
note = float(input("Note sur 20 : "))
if note == 0.0:
    print("C'est en dessous de la moyenne")
    print("... lamentable!")
elif note == 20.0:
    print("J'ai la moyenne")
    print("C'est même excellent !")
elif 0 < note < 10:    # ou bien : elif 0.0 < note < 10.0:
    print("C'est en dessous de la moyenne")
elif note >= 10.0 and note < 20.0:   # ou bien : elif 10.0 <= note < 20.0:
    print("J'ai la moyenne")
else:
    print("Note invalide !")
print("Fin du programme")
```

Exercise 1 : Condition sur le jour de travail

Si aujourd'hui est lundi alors je dois aller travailler, mais si c'est dimanche alors je peux rester faire la grasse matinée. Pour pouvoir accomplir ce genre de choses en Python, on fait appel à des expressions booléennes qui ne peuvent revêtir que deux possibilités - ou bien l'expression est vraie ou bien elle est fausse - et à la syntaxe if condition : qui permet de contrôler le flux du programme grâce à ces valeurs booléennes.

```
day_week = ["Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi",
            "Samedi", "Dimanche"]
today = input("Aujourd'hui est: ")

if *condition vraie*:  # Quelle est la condition vraie dans ce cas?
    print("Je dors le matin!")
else:
    print("Je travail le matin!")

print("Fin du programme")
```

Indication. Dans la **condition vraie**, utilisez l'opérateur logique `in` pour tester les éléments de la liste `day_week`.

```
day_week = ["Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche"]
today = input("Aujourd'hui est: ")
if today in day_week:
    if today == day_week[-1]:  # Quelle est la condition vraie dans ce cas?
        print("Je dors le matin!")
    else:
        print("Je travail le matin!")

print("Fin du programme")
```

Solution.

2 Les boucles

2.1 L'instruction `while`

Syntaxe

```
while expression:           # ne pas oublier le signe de ponctuation ':'
    "bloc d'instructions"   # attention à l'indentation (1 Tab ou 4 * Espaces)
# suite du programme
```

— Si l'expression est vraie (`True`) le bloc d'instructions est exécuté, puis l'expression est à nouveau évaluée.

— Le cycle continue jusqu'à ce que l'expression soit fausse (`False`) : on passe alors à la suite du programme.

```
# initialisation de la variable de comptage
compteur = 0
while compteur < 5:
    # ce bloc est exécuté tant que la condition (compteur < 5) est vraie
    print(compteur)
    compteur = compteur + 1    # incrémentation du compteur,  compteur = compteur + 1
print(compteur)
print("Fin de la boucle")
```

Exemple 1 : un script qui compte de 1 à 4.

```
compteur = 1          # initialisation de la variable de comptage
while compteur <= 10:
    # ce bloc est exécuté tant que la condition (compteur <= 10) est vraie
    print(compteur, '* 8 =', compteur*8)
    compteur += 1    # incrémentation du compteur, compteur = compteur + 1
print("Et voilà !")
```

Exemple 2 : Table de multiplication par 8.

```
import time      # importation du module time
quitter = 'n'    # initialisation
while quitter != 'o':
    # ce bloc est exécuté tant que la condition est vraie
    # strftime() est une fonction du module time
    print('Heure courante ', time.strftime('%H:%M:%S'))
    quitter = input("Voulez-vous quitter le programme (o/n) ? ")
print("A bientôt")
```

Exemple 3 : Affichage de l'heure courante.

2.2 L'instruction `for`

Syntaxe

```
for élément in séquence :      # ne pas oublier le signe de ponctuation ':'
    "bloc d'instructions"      # attention à l'indentation (1 Tab ou 4 * Espaces)
# suite du programme
```

Les éléments de la séquence sont issus d'une chaîne de caractères ou bien d'une liste.

```
chaine = 'Bonsoir'
for lettre in chaine:  # lettre est la variable d'itération
    print(lettre)
print("Fin de la boucle")
```

Exemple 1 : séquence de caractères.

La variable lettre est initialisée avec le premier élément de la séquence ('B'). Le bloc d'instructions est alors exécuté.

Puis la variable lettre est mise à jour avec le second élément de la séquence ('o') et le bloc d'instructions à nouveau exécuté...

Le bloc d'instructions est exécuté une dernière fois lorsqu'on arrive au dernier élément de la séquence ('r').

Fonction `range()`. L'association avec la fonction `range()` est très utile pour créer des séquences automatiques de nombres entiers :

```
for i in range(1, 5):
    print(i)
print("Fin de la boucle")
```

Exemple 2 : Table de multiplication. La création d'une table de multiplication paraît plus simple avec une boucle `for` qu'avec une boucle `while` :

```
for compteur in range(1,11):
    print(compteur, '* 8 =', compteur*8)
print("Et voilà !")
```

Exemple 3 : calcul d'une somme. Soit, par exemple, l'expression de la somme suivante :

$$s = \sum_{i=0}^{100} \sqrt{\frac{i\pi}{100}} sin(\frac{i\pi}{100})$$

```
from math import sqrt, sin, pi
s = 0.0 # # intialisation de s
for i in range(101):
    s+= sqrt(i * pi/100) * sin(i * pi/100)    # équivalent à s = s + sqrt(x) * sin(x)
# Affichage de la somme
print(s)
```

Exercise 2 : produit de Wallis

Calculer π avec le produit de Wallis

$$\frac{\pi}{2} = \prod_{i=1}^{p} \frac{4i^2}{4i^2 - 1}$$

```
# %load wallis.py
from math import pi

my_pi = 1.  # intialisation
p = 100000
for i in range(1, p):
    my_pi *= 4 * i ** 2 / (4 * i ** 2 - 1.)  # implémentation de la formule de Wallis

my_pi *= 2       # multiplication par 2 de la valeur trouvée

print("La valeur de pi de la bibliothèque 'math': ", pi)
print("La valeur de pi calculer par la formule de Wallis: ", my_pi)

print("La différence entre les deux valeurs:", abs(pi - my_pi))
# la fonction abs() donne la valeur absolue
```

Solution.

2.3 Compréhensions de listes

Les compréhensions de listes fournissent un moyen de construire des listes de manière très concise. Une application classique est la construction de nouvelles listes où chaque élément est le résultat d'une opération appliquée à chaque élément d'une autre séquence ; ou de créer une sous-séquence des éléments satisfaisant une condition spécifique.

Par exemple, supposons que l'on veuille créer une liste de carrés, comme :

```
squares = []
for x in range(10):
    squares.append(x**2)
```

```
squares
```

Notez que cela crée (ou remplace) une variable nommée `x` qui existe toujours après l'exécution de la boucle. On peut calculer une liste de carrés sans effet de bord avec :

```
squares = [x**2 for x in range(10)]
squares
```

qui est plus court et lisible.

Une compréhension de liste consiste à placer entre crochets une expression suivie par une clause `for` puis par zéro ou plus clauses `for` ou `if`. Le résultat est une nouvelle liste résultat de l'évaluation de l'expression dans le contexte des clauses `for` et `if` qui la suivent. Par exemple, cette compréhension de liste combine les éléments de deux listes s'ils ne sont pas égaux :

```
combs = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
```

```
combs
```

et c'est équivalent à :

```
combs = []
for x in [1,2,3]:
    for y in [3,1,4]:
        if x != y:
            combs.append((x, y))
```

```
combs
```

Note
Notez que l'ordre des instructions `for` et `if` est le même dans ces différents extraits de code.

2.4 L'instruction break

L'instruction break provoque une sortie immédiate d'une boucle `while` ou d'une boucle `for`.

Dans l'exemple suivant, l'expression `True` est toujours ... vraie : on a une boucle sans fin.

L'instruction `break` est donc le seul moyen de sortir de la boucle.

```
import time     # importation du module time
while True:
    # strftime() est une fonction du module time
    print('Heure courante ', time.strftime('%H:%M:%S'))
    quitter = input('Voulez-vous quitter le programme (o/n) ? ')
    if quitter == 'o':
        break
print("A bientôt")
```

Exemple : Affichage de l'heure courante.

Note
Si vous connaissez le nombre de boucles à effectuer, utiliser une boucle `for`. Autrement, utiliser une boucle `while` (notamment pour faire des boucles sans fin).

3 Les fonctions

Nous avons déjà vu beaucoup de fonctions : `print()`, `type()`, `len()`, `input()`, `range()`...

Ce sont des fonctions pré-définies (Fonctions natives).

Nous avons aussi la possibilité de créer nos propres fonctions !

3.1 Intérêt des fonctions

Une fonction est une portion de code que l'on peut appeler au besoin (c'est une sorte de sous-programme).

L'utilisation des fonctions évite des redondances dans le code : on obtient ainsi des programmes plus courts et plus lisibles.

Par exemple, nous avons besoin de convertir à plusieurs reprises des degrés Celsius en degrés Fahrenheit :

$$T_F = T_C \times 1,8 + 32$$

```
print(100 * 1.8 + 32.0)
```

```
print(37.0 * 1.8 + 32.0)
```

```
print(233.0 * 1.8 + 32.0)
```

La même chose en utilisant une fonction :

```
def fahrenheit(degre_celsius):
        """
        Conversion degré Celsius en degré Fahrenheit
        """
        print(degre_celsius * 1.8 + 32.0)
```

```
fahrenheit(100)
```

```
fahrenheit(37)
```

```
temperature = 220
fahrenheit(temperature)
```

3.2 L'instruction def

Syntaxe

```
def nom_de_la_fonction(parametre1, parametre2, parametre3, ...):
    """
    Documentation
    qu'on peut écrire
    sur plusieurs lignes
    """     # docstring entouré de 3 guillemets (ou apostrophes)

    "bloc d'instructions"     # attention à l'indentation

    return resultat            # la fonction retourne le contenu de la variable resultat
```

```
def mapremierefonction():          # cette fonction n'a pas de paramètre
    """
    Cette fonction affiche 'Bonjour'
    """
    print("Bonjour")
    return                          # cette fonction ne retourne rien ('None')
    # l'instruction return est ici facultative
```

Exemple : ma première fonction.

```
mapremierefonction()
```

```
help(mapremierefonction)
```

4 Les Scripts

Commençons par écrire un script, c'est-à-dire un fichier avec une séquence d'instructions à exécuter chaque fois que le script est appelé. Les instructions peuvent être, par exemple, copié-collé depuis une **cellule code** dans votre notebook (mais veillez à respecter les règles d'indentation !).

L'extension pour les fichiers Python est `.py`. Écrivez ou copiez et collez les lignes suivantes dans un fichier appelé `test.py`

```
chaine = 'Bonsoir'
for lettre in chaine:  # lettre est la variable d'itération
    print(lettre)
```

Exécutons maintenant le script de manière interactive, à l'intérieur de l'interpréteur Ipython (cellule code du notebook). C'est peut-être l'utilisation la plus courante des scripts en calcul et simulation scientifique.

Note
Dans la cellule *code* (Ipython), la syntaxe permettant d'exécuter un script est `%run script.py`. Par exemple :

```
%run test.py
```

```
chaine
```

La syntaxe permettant de charger le contunu d'un script dans dans une cellule code est `%load script.py`. Par exemple :

```python
# %load test.py
chaine = 'Bonsoir'
for lettre in chaine:  # lettre est la variable d'itération
    print(lettre)
```

5 Lectures complémentaires

— Documentation Python 3.6
— Apprendre à programmer avec Python, par Gérard Swinnen
— Think Python, par Allen B. Downey