Les structures de contrôle

Principe

Nous avons précédemment évoqué que, lors de l'exécution d'un programme en Python, l'ordinateur exécute **séquentiellement** (c'est-à-dire dans l'ordre, de manière successive) les instructions écrites sur chacune des lignes.

Une structure de contrôle permet toutefois de rediriger l'exécution du programme vers une partie antérieure ou postérieure du code : c'est indispensable pour réaliser un programme complexe (il faudrait sinon écrire toutes les instructions « à la main »).

Les principales structures de contrôle disponibles en Python (et dans tous les langages) sont :

- l'appel de fonction ;
- l'alternative (`if`) ;
- la boucle non bornée (`while`) ;
- la boucle bornée (`for`) ;

Il en existe d'autres en Python (`break`, `pass`, gestion des exceptions) et dans d'autres langages (`goto`, `until`, `switch` ou `case`, *etc.*).

1 L'appel de fonction

Une fonction est une portion de code qui effectue un traitement spécifique et qui peut être utilisée partout dans la suite du programme (nous verrons plus tard qu'une fonction peut en outre renvoyer un « résultat »).
En python, la déclaration d'une fonction se fait à l'aide du mot-clef `def`.
Les instructions à exécuter forment le corps de la fonction : elles doivent être indentées, c'est-à-dire en retrait. On utilise pour cela la touche tabulation du clavier, qui insère quatre espaces.

Déclaration et appel !

Il ne faut pas confondre la déclaration d'une fonction et l'appel de la fonction. Une déclaration de fonction, c'est l'équivalent d'une recette de cuisine sur papier. Pour goûter la recette, il faut la cuisiner. Pour tester la fonction, il faut l'appeler !

```
1 def hello():
2     print("Hello world!")
```

Le script ci-dessus n'affiche rien. En effet, si la fonction `hello` a bien été définie, le code **ne contient aucun appel** à celle-ci. On doit donc modifier le script comme suit.

```
1 def hello():
2     print("Hello world!")
3
4 hello()
```

Il est possible de paramètrer une fonction, c'est-à-dire de lui préciser la valeur d'un ou de plusieurs paramètres à l'appel de la fonction.

Par exemple, le code ci-dessous produit les affichages à droite.

```
def hello(name):
    print("Hello ", name, "!")

hello("Alan")
hello("Barbara")
```

```
Hello Alan!
Hello Barbara!
```

Par simplification, on utilise souvent le nom « paramètre » pour deux concepts différents :

- le nom utilisé dans la définition d'une fonction pour désigner une valeur qui ne sera connue qu'au moment de chaque appel, on l'appelle **paramètre formel** ;
- la valeur concrète associée au paramètre formel au moment d'un appel de fonction, on l'appelle **paramètre effectif ou réel**.

Exercice 1

On définit le script ci-dessous :

```
def test(x, y):
    somme = x + y
    print(somme)

a = 5
b = 4
test(a, b)
```

1. Quel est le nom de la fonction ?
2. Quels sont les paramètres formels de la fonction ?
3. Quels sont les paramètres effectifs de la fonction ?
4. Quelles sont les variables locales à la fonction ? (c'est-à-dire qui n'existent qu'à l'intérieur de la fonction) ?
5. Qu'affiche ce programme ?

2 L'alternative (`if`)

L'alternative (appelée aussi test conditionnel) exécute un bloc d'instructions si (et seulement si) une condition donnée a pour valeur `True`. La syntaxe en Python est la suivante :

```
if condition :
    instruction_1
    instruction_2
    etc.
```

Toutes les instructions à exécuter, dans le cas où la condition est vraie, doivent être indentées.
La condition peut prendre la forme d'un test d'égalité (`==`) ou de non-égalité (`!=`), d'un test de comparaison (`<`, `>`, `<=`, `>=`), d'une variable booléenne ou encore d'une expression booléenne.

Exercice 2

Qu'affiche le script ci-dessous ? (à faire sur papier d'abord)

```
1  a = 5
2  b = 7
3  v = False
4  if a > b or v :
5      a = 8
6      v = True
7  v = not(v)
8  if a > b or v :
9      b = 9
10 print(a, b, v)
```

Le mot-clef `else` permet de préciser les instructions à exécuter dans le cas où la condition a pour valeur `False`.
Les deux syntaxes ci-dessous sont donc équivalentes.

```
if condition :
    instructions_1
if not(condition) :
    instructions_2
```

```
if condition :
    instructions_1
else :
    instructions_2
```

Le mot-clef `elif` (contraction de `else` et `if`) permet d'exécuter des instructions dans le cas où une première condition est fausse et où une seconde est vraie.
Les deux syntaxes ci-dessous sont donc équivalentes.

```
if condition_1 :
    instructions_1
else :
    if condition_2 :
        instructions 2
```

```
if condition_1 :
    instructions_1
elif condition_2 :
    instructions_2
```

Exercice 3

Les amis de Bob décident de faire un intervention parce que celui-ci préfère jouer à la console plutôt que réviser ses cours de NSI le week-end. Bob, très concerné, finit par admettre qu'ils ont raison et leur annonce qu'il va désormais répartir son emploi du temps du week-end comme suit :

- Bob est en mode révision NSI de 10h à 14h
- Bob peut jouer à la console le reste du temps (ou vivre sa vie de Bob comme manger, dormir, etc. bien sûr)

Ses amis décident de coder un programme pour suivre ce que fait Bob. Ce programme affichera le mode, soit "NSI", soit "console" en fonction de l'heure. Mais comme ils ne révisent pas assez non plus à force de surveiller Bob, ils n'arrivent pas à finir. Voici où ils en sont :

```
1  def bob_en_weekend(heure):
2      # 'mode' est une chaîne de caractères :
3      # soit "NSI", soit "console"
4      if heure > 0 and heure < 10:
5          mode = "console"
6      elif heure < 14:
7          mode = ...
8      elif heure > 14 and heure < 24:
9          mode = ...
10
11     print(mode)
```

1. Compléter les lignes 7 et 9.
2. Comment pourrait-on écrire le programme différemment pour n'avoir que deux lignes de tests conditionnels ?

Finalement, Bob en a marre et décide qu'il veut aussi jouer à la console le mercredi et qu'il se considérera donc en mode week-end ce jour-là. Il change alors un peu le programme en ajoutant une nouvelle fonction :

```
12 def mode_bob(jour, heure):
13     # on admet que 'jour' est une chaîne de caractère :
14     # "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi" ou "dimanche"
15     if jour == "samedi" ...
16         ...
17     else:
18         mode = "NSI"
19         print(mode)
```

3. Compléter cette fonction pour sauver Bob de l'enfer des révisions.

3 La boucle non bornée

En Python, la boucle non bornée est la boucle `while` : elle exécute un bloc d'instructions tant qu'une condition donnée a pour valeur `True`. La syntaxe en Python est la suivante :

```
while condition :
    instruction_1
    instruction_2
    etc.
```

Toutes les instructions à exécuter, dans le cas où la condition est vraie, doivent être indentées.
On utilise une boucle non bornée lorsque le nombre de tours de boucle est *a priori* inconnu.

Exercice 4

On considère la fonction `foo` suivante.

```
1  def foo(n):
2      p = 1
3      while p<n :
4          p = p*2
5      print(p/2)
```

1. (a) Dérouler à la main le programme lors de l'appel `foo(100)`.
   (b) Combien de tours de boucle sont effectués ?
   (c) Quelle est l'affichage produit à la ligne 5 ?
2. De manière générale, quel est le nombre affiché produit lors de l'appel `foo(n)` ?

Exercice 5

Pour chacun des scripts ci-dessous, déterminer les affichages produits.

```
1  x = 2
2  while x>100 :
3      x = 2*x
4      x = x-1
5  print(x)
```

```
1  x = 2
2  while x<100 :
3      x = 2*x
4      x = x-1
5  print(x)
```

```
1  x = 1
2  while x<100 :
3      x = 2*x
4      x = x-1
5  print(x)
```

⚠ Une boucle non bornée peut ne jamais se terminer !

Cette situation peut déboucher sur un « plantage » de l'IDE (*Integrated Development Environment* – environnement de développement intégré), voire du système d'exploitation.

Il est donc indispensable de s'assurer de la terminaison des boucles non bornées, c'est-à-dire de vérifier que la condition suivant le mot-clé `while` finit par être égale à `False`.

Exercice 6

Pour quelles valeurs de `m` la fonction `bar` ci-dessous se termine-t-elle ?

```
1  def bar(m):
2      x = m
3      while x>0 and x<100:
4          x = x-1
```

4 La boucle bornée

La plus simple des boucles bornées consiste à répéter un certain nombre de fois un bloc d'instructions.
Toutefois, cette boucle n'est pas disponible en l'état dans le langage Python.
Mais il est possible de faire prendre à une variable les différentes valeurs contenus dans un itérable .

Définition

Un itérable est un objet composé de plusieurs éléments et possédant un itérateur, c'est-à-dire d'une méthode permettant de passer d'un élément au suivant selon un ordre déterminé.
Un des itérables les plus simples est fourni par la fonction `range(n)` qui permet de parcourir les entiers naturels de `0` jusqu'à `n-1`.

Exercice 7

On considère le script suivant.

```
for i in range(n):
    print(2*i)
```

Quels affichages produisent son exécution ? Combien de tours de boucle ont été effectués ?

Exercice 8

On a utilisé dans la fonction ci-dessous une boucle `while`.

```
def truc(n):
    x = 1
    k = 1
    while k<n:
        x = 2*x
        k = k+1
    print(x)
```

1. Quelle est la valeur affichée lors de l'appel `truc(5)` ?
2. Proposer une version de la fonction `truc` utilisant une boucle `for`.

Exercice 9

Compléter la fonction suivante pour qu'elle affiche la somme de tous les entiers naturels inférieurs ou égaux au paramètre `n`.

```
def somme(n):
    s =
    for
        s = s+k
    print(s)
```

Exercice 10

En théorie, une boucle bornée se termine toujours (contrairement aux boucles non bornées, d'où la terminologie).
En pratique, cela n'est par forcément le cas, comme dans l'exemple (stupide) ci-dessous.

```
1 n = 10**12
2 k = 0
3 for i in range(n):
4     k = k+1
5 print(k)
```

Quelle doit être la valeur affichée par l'instruction ligne 5 ? Combien de tours de boucle sont nécessaires ?