Informatique
TP2 : Structures de contrôles, algorithmes
CPP 1A

Djamel Aouane, Frederic Devernay, Matthieu Moy

Mars - avril 2015

1 Boucles `while`

Nous avons vu en cours comment écrire une multiplication en utilisant des additions successives. L'algorithme est rappelé ici :

```
x = 4
y = 5

print("Avant multiplication : x =", x, "et y =", y)

resultat = 0
while y > 0:
    resultat = resultat + x
    y = y - 1
    print("Dans la boucle, resultat =", resultat)

print("Résultat :", resultat)
```

Exercice 1 (Exécution du programme) *Télécharger ce programme depuis la page du cours (fichier* ***`multiplication.py`****) et ouvrez-le dans Spyder (depuis Spyder, menu « Fichier », « Ouvrir »).*

Exécutez le programme (bouton ▶ « Exécuter le fichier » dans la barre d'outils). Essayez avec d'autres valeurs pour `x` *et* `y` *et vérifier que l'algorithme fait bien une multiplication.*

La sortie du programme devrait ressembler à ceci :

```
Avant multiplication : x = 4 et y = 5
Dans la boucle, resultat = 4
Dans la boucle, resultat = 8
Dans la boucle, resultat = 12
Dans la boucle, resultat = 16
Dans la boucle, resultat = 20
Résultat : 20
```

Pour pouvoir observer chaque pas de l'exécution, nous avons utilisé l'instruction

```
print("Dans la boucle, resultat =", resultat)
```

dans le corps de la boucle.

Exercice 2 (Mise en commentaire) *Mettez cette ligne en commentaire (i.e. ajoutez un caractère* `#` *en début de ligne) et ré-exécutez le programme. L'affichage disparait. Décommentez cette ligne (i.e. revenez au point de départ) et vérifiez que l'affichage revient.*

Pour l'instant, l'instruction `print("Dans la boucle, resultat =", resultat)` fait partie du corps de la boucle, car il fait partie du bloc indenté.

Exercice 3 (Boucle while et indentation) *Essayez de désindenter cette ligne, c'est à dire de supprimer les 4 espaces en début de ligne comme ceci :*

```
while y > 0:
    resultat = resultat + x
    y = y - 1
print("Dans la boucle, resultat =", resultat)
```

Exécutez le programme. La sortie devrait ressembler à ceci :

```
Avant multiplication : x = 4 et y = 5
Dans la boucle, resultat = 20
Résultat : 20
```

L'affichage est maintenant fait une seule fois, après la sortie de boucle.

Exercice 4 (Erreurs avec l'indentation) *Essayez maintenant d'indenter cette ligne* `print` *avec seulement 3 espaces, comme ceci :*

```
while y >= 0:
    resultat = resultat + x
    y = y - 1
   print("Dans la boucle, resultat =", resultat)
```

Python considère maintenant ce programme comme incorrect et refuse de l'exécuter.
Revenez à la version d'origine (indentation, avec 4 espaces).

Exercice 5 (Minimisation du nombre d'additions) *Si* `y` *est plus grand que* `x`*, calculer* `x` $+$ `x` $+ \cdots +$ `x` *(*`y` *fois) est inefficace, et il vaut mieux calculer* `y` $+$ `y` $+ \cdots +$ `y` *(*`x` *fois). Optimisez votre programme en ajoutant avant la boucle* `while` *quelques lignes signifiant « si* `y` *est plus grand que* `x`*, alors échanger* `x` *et* `y` *».*

1.1 Déboguage avec Spyder

Dans la section précédente, nous avons utilisé l'instruction `print` pour observer l'exécution du programme. Une autre méthode, plus propre, est d'utiliser un débogueur, qui permet d'observer une exécution du programme sans le modifier.

Pour activer le débogueur, choisissez menu « Déboguer », « Déboguer », ou bien cliquez sur le bouton correspondant dans la barre de boutons (« Déboguer le script » en forme de bouton « play/pause », ou Control+F5). L'exécution est prête, mais rien n'a encore été exécuté : à nous d'exécuter les instructions une par une.

La barre de boutons du haut contient entre autres les actions :

– « Exécuter la ligne en cours » pour exécuter jusqu'à la ligne suivante du programme pour continuer l'exécution sans déboguer

– « Avancer dans la fonction ... » pour exécuter l'instruction suivante. Si l'instruction courante est un appel de fonction, le débogueur « entre » dans la fonction
– « Exécuter jusqu'au retour de la fonction ou méthode » pour exécuter jusqu'à sortir de la fonction courante (très utile si on est entré par erreur dans une fonction)

En choisissant l'onglet « Explorateur de variables » du panneau en haut à droite de la fenêtre Spyder, on peut voir les variables du programme, leur type et leur valeur.

Exercice 6 (Déboguage) *Exécutez pas à pas le programme avec « Exécuter la ligne en cours », jusqu'à sa fin, et observez les changements des valeurs* `x`, `y` *et* `resultat`.

1.2 Modification de l'algorithme

Exercice 7 (Puissance par multiplication successives) *En vous partant de cet algorithme, écrivez un algorithme qui calcule la valeur x^y par multiplications successives*[1]*. Testez cet algorithme en affichant son résultat avec* `print` *et/ou le débogueur.*

2 Maximum d'un ensemble de nombres

Exercice 8 (Maximum de deux nombres) *Écrivez un algorithme de calcul du maximum de deux nombres. Votre programme doit ressembler à ceci :*

```
a = 12
b = 14

# ... calcul dans la variable 'max' ...

print("Le maximum est :", max)
```

Vérifiez que votre programme fonctionne dans tous les cas (`a` $<$ `b`, `a` $=$ `b` *et* `a` $>$ `b`*).*

Exercice 9 (Maximum de trois nombres) *Écrivez un algorithme de calcul du maximum de trois nombres. Votre programme doit ressembler à ceci :*

```
a = 12
b = 14
c = 13

# ... calcul dans la variable 'max' ...

print("Le maximum est :", max)
```

On peut remarquer que l'algorithme se complique assez vite. Calculer le maximum de 4 ou 5 nombres risque d'être difficile si on ne s'organise pas un peu ...

Une solution est de parcourir l'ensemble des nombres à considérer, et pour chaque nombre de regarder s'il est plus grand que le plus grand nombre considéré auparavant.

Nous ne savons pas encore stocker n nombres dans des variables, donc nous allons les demander à l'utilisateur au fur et à mesure.

Exercice 10 (Maximum d'une suite de nombres) *Écrivez un algorithme de calcul du maximum de n nombres, en demandant à l'utilisateur d'entrer une suite de nombre au clavier. L'utilisateur peut entrer une valeur négative pour terminer. Votre programme peut ressembler à ceci :*

1. On pourrait bien sûr utiliser l'opérateur `**`, mais ce n'est pas le but de l'exercice

```
print("Entrez la premiere valeur :")
x = int(input())
# ...

while x >= 0:
    # ...
    print("Entrez la valeur suivante :")
    x = int(input())

print("Le maximum est :", max)
```

3 Définition et utilisations de fonctions

Une portion de code utilisée à plusieurs endroits du programme peut être écrite une fois dans une *fonction*, puis utilisée à plusieurs endroits en *appelant* la fonction. Par exemple :

```
def annoncer_train(d, h):
    print("Le train à destination de", d, "partira a", h)

annoncer_train("Grenoble", "9h")
annoncer_train("Paris", "10h15")
```

Comme en mathématique, une fonction peut renvoyer une valeur. Par exemple, la fonction qui calcule la longueur de l'hypoténuse d'un triangle rectangle en fonction de la longueur des deux autres côtés peut s'écrire :

```
def hypotenuse(x, y):
    return (x ** 2 + y ** 2) ** 0.5
```

L'instruction `return` a deux effets :
- Elle arrête l'exécution de la fonction
- Elle renvoie la valeur à l'instruction qui a appelé la fonction (par exemple : `resultat = hypotenuse(x, y)`).

Exercice 11 (Utilisation d'une fonction) *Écrivez un programme qui définit la fonction* `hypotenuse` *ci-dessus et qui l'utilise pour calculer* $\sqrt{3^2 + 4^2}$*, et vérifiez que le résultat est correct.*

Exercice 12 (Fonction sans return) *Essayez de modifier la définition de la fonction en supprimant le mot clé* `return` *:*

```
def hypotenuse(x, y):
    (x ** 2 + y ** 2) ** 0.5
```

Exécutez à nouveau votre programme. Que se passe-t-il ?

Exercice 13 (Fonction "maximum de deux nombres") *Reprenez l'algorithme de calcul du maximum de deux nombres écrit ci-dessus, et faites-en une fonction* `max2(a, b)`*. Vous aurez sans doute besoin de deux instructions* `return`*.*

Remarque : avec cette fonction, calculer le maximum de 3 nombres devient plus facile ! On peut écrire :

```
def max3(a, b, c):
    return max2(a, max2(b, c))
```

4 Programmes multi-fichiers

Quand on écrit des fonctions qui sont destinées à être utilisées dans plusieurs programmes, ou lorsqu'on veut séparer les parties d'un programme (par exemple la partie qui fait le calcul, la partie qui fait l'affichage, et la partie qui faite le test), on peut séparer le programme en plusieurs fichiers .py : les *modules* qui ne contiennent que des fonctions, et les *programmes* qui utilisent les modules.

En fait, Python est fourni avec un ensemble de modules standard, qui contiennent déjà beaucoup de fonctions. Par exemple, le module `math` contient une fonction `sqrt` (square root) que l'on peut utiliser au lieu de `** 0.5` pour calculer une racine carrée. Pour l'utiliser, il faut d'abord écrire (en début de programme) **`import`** `math`, puis appeler la fonction `math.sqrt` comme ceci :

```
import math

def hypotenuse(x, y):
    return math.sqrt(x ** 2 + y ** 2)
```

Le principe est le même pour les fonctions définies par l'utilisateur. Supposons qu'on ait mis la fonction hypotenuse ci-dessus dans un fichier `mesmaths.py`, ont peut alors écrire un programme nommé par exemple `mesmaths_tests.py` (dans le même répertoire) contenant :

```
import mesmaths
print("mesmaths.hypotenuse(3, 4) =", mesmaths.hypotenuse(3, 4))
```

On exécute le fichier avec F5 dans Spyder (attention, il faut sauvegarder tous les fichiers du programme et sélectionner la fenêtre contenant `mesmaths_tests.py`). `mesmaths.hypotenuse` fait référence à la fonction `hypotenuse` du module `mesmaths` (comme pour `math.sqrt`).

À noter qu'on peut également écrire la *documentation* de chaque fonction en mettant une chaîne de caractères au début de la fonction :

```
def hypotenuse(x, y):
    """Retourne l'hypotenuse d'un triangle rectangle de côtés x et y"""
    return math.sqrt(x ** 2 + y ** 2)
```

La documentation d'une fonction `f` peut être affichée avec `help(f)` (essayez avec `math.sqrt`). Pensez toujours à utiliser des noms de fonctions explicites et à les documenter.

Exercice 14 (Multi-fichiers) *Que se passe-t-il si on met des instructions qui ne sont pas des définitions de fonctions dans le module (par exemple un* `print`*) ? Définissez dans* `mesmaths_tests.py` *une autre fonction nommée aussi* `hypotenuse` *renvoyant un résultat incorrect. Vérifiez que* `mesmaths.hypotenuse` *fait toujours référence à la version écrite dans* `mesmaths.py`*. Complétez* `mesmaths.py` *en y ajoutant les fonctions (documentées) multiplication et puissance, et écrivez les tests correspondants.*

Bien sûr, on peut aussi utiliser **`import`** pour appeler des fonctions dans des modules fournis avec Python (qu'on appelle la *bibliothèque standard*).

Exercice 15 (Bibliothèque standard) *Modifiez la fonction* `hypotenuse` *pour remplacer* `** 0.5` *par un appel à la fonction* `math.sqrt`*. Vous aurez besoin d'utiliser* **`import`** `math` *en début de fichier, puis d'appeler* `math.sqrt(...)`*.*

5 Si le temps le permet ...

Le programme `multiplication.py` que nous avons utilisé au début du TP n'est pas très bien structuré :

- Les valeurs de `x` et `y` sont codées en dur au début du programme (`x = 4` et `y = 5`). On ne peut pas utiliser notre algorithme pour calculer autre chose que 4×5 sans modifier le fichier.
- L'algorithme a proprement parler est mélangé avec l'affichage du résultat.

Il serait beaucoup plus propre d'avoir une fonction `multiplication`, prenant deux paramètres `x` et `y` et renvoyant le résultat.

Exercice 16 (Structuration en fonction) *Reprenez le programme de la multiplication écrit au début du TP, placez l'algorithme de multiplication à proprement parler dans une fonction et appelez cette fonction. Le programme obtenu doit faire exactement la même chose que le programme d'origine. La fonction* `multiplication` *ne doit s'occuper que du calcul, et pas des affichages. Vous pouvez supprimer l'instruction* **`print`** *qui se trouve à l'intérieur de la boucle et qui ne sert qu'a observer l'exécution du programme.*

Il reste un problème : la fonction est déclarée dans le même fichier que le code qui l'utilise. Il n'est pas (encore) possible pour un autre programme d'utiliser notre fonction en utilisant **`import`**.

Exercice 17 (Découpage en plusieurs fichiers) *Découpez votre programme en deux fichiers : l'un doit uniquement contenir la définition de la fonction* `multiplication`*, et l'autre doit s'occuper des affichages, puis appeler la fonction (il est nécessaire d'utiliser* **`import`** *pour avoir le droit d'utiliser la fonction).*

Exercice 18 (Multiplication dans l'Égypte antique) *L'algorithme de produit proposé multiplie x par y en additionnant x y fois.*

Les Égyptiens, qui ne connaissaient pas la multiplication, avaient un algorithme beaucoup plus économique : à chaque pas un des opérandes (y) est doublé, l'autre (x) divisé par deux. Avant cette opération, si x n'est pas pas pair, alors on ajoute y au résultat.

Par exemple :

$$25 \times 7 = 12 \times 14 + 7 = 6 \times 28 + 7 = 3 \times 56 + 7 = 1 \times 112 + 56 + 7 = 175$$

Écrire une fonction qui calcule le produit de deux entiers selon la méthode des égyptiens, en n'utilisant que la multiplication par deux et la division par deux.

Comparer le nombre d'itérations de cet algorithme avec celui du premier algorithme. De quoi dépend ce nombre d'itérations ? Sauriez-vous l'exprimer comme une fonction de x et y ?

6 Solution des exercices

Exercice 5 :

```
x = 4
y = 5

print("Avant multiplication : x =", x, "et y =", y)

# Echange des variables x et y si nécessaire
if y > x:
    tmp = x
```

```
    x = y
    y = tmp

resultat = 0
while y > 0:
    resultat = resultat + x
    y = y - 1
    print("Dans la boucle, resultat =", resultat)

print("Résultat :", resultat)
```

Exercice 7 : Il suffit de remplacer le `+` par un `*`, et d'initialiser le résultat au neutre de la multiplication (1) au lieu de 0.

```
x = 4
y = 5

print("Avant multiplication : x =", x, "et y =", y)

resultat = 1
while y > 0:
    resultat = resultat * x
    y = y - 1
    print("Dans la boucle, resultat =", resultat)

print("Résultat :", resultat)
```

Exercice 8 :

```
a = 12
b = 14

if a > b:
    max = a
else:
    max = b

print("Le maximum est :", max)
```

Exercice 9 : Attention à l'indentation, nous avons bien un **`if/else`** qui contient lui-même un **`if/else`** dans sa branche **`if`** et un autre **`if/else`** dans sa branche **`else`**.

```
a = 1
b = 5
c = 3

if a > b:
    if a > c:
        max = a
    else:
        max = c
else:
    if b > c:
        max = b
    else:
        max = c

print("Le maximum est :", max)
```

Exercice 10 :

```
print("Entrez la premiere valeur :")
x = int(input())
max = x

while x >= 0:
    if x > max:
        max = x
    print("Entrez la valeur suivante :")
    x = int(input())

print("Le maximum est :", max)
```

Exercice 11 :

```
def hypotenuse(x, y):
    """calcul de l'hypotenuse d'un triangle rectangle"""
    return (x ** 2 + y ** 2) ** 0.5

print(hypotenuse(3, 4))
```

Exercice 12 : la fonction ne renvoie plus rien [2], le programme ne marche plus. Une fonction sans return est utilisable comme une instruction, mais pas comme une expression.

Exercice 13 :

```
def max2(a, b):
    if a > b:
        return a
    else:
        return b

def max3(a, b, c):
    return max2(a, max2(b, c))

print("Le maximum est :", max2(10, 12))
print("Le maximum est :", max3(101, 12, 14))
```

Exercice 14 : **Contenu du fichier `mesmath.py` :**

```
import math

def hypotenuse(x, y):
    """calcul de l'hypotenuse d'un triangle rectangle"""
    return math.sqrt(x ** 2 + y ** 2)

print("mesmaths.py a ete lu")
```

Contenu du fichier `mesmath_test.py` :

```
import mesmaths

def hypotenuse(x, y):
    return x ** 2 + y ** 2

print("mesmaths.hypotenuse(3, 4) =", mesmaths.hypotenuse(3, 4))
print("hypotenuse(3, 4) =", hypotenuse(3, 4))
help(mesmaths.hypotenuse)
```

le code en dehors des fonctions est exécuté lors de l'importation du module. `hypotenuse` et `math.hypotenuse` sont bien deux fonctions distinctes.

Exercice 15 :

```
import math

def hypotenuse(x, y):
    """calcul de l'hypotenuse d'un triangle rectangle"""
    return math.sqrt(x ** 2 + y ** 2)

print(hypotenuse(3, 4))
```

Exercice 17 : **Contenu du fichier fichier `module_multiplication.py` :**

```
def multiplication(x, y):
    resultat = 0
    while y > 0:
          resultat = resultat + x
          y = y - 1
          # print("Dans la boucle , resultat = " , resultat )
    return resultat
```

Contenu du fichier `test_multiplication.py` :

```
import module_multiplication

x = 4
y = 5
print("Avant multiplication : x = ", x, " et y = ", y)
print("Résultat :",
      module_multiplication.multiplication(x, y))
```

2. Techniquement, elle renvoie la valeur `None` mais on ne peut rien faire avec.