Fonctions

Introduction

En Python 3. comme dans la plupart des langages, on peut définir soi-même de nouvelles fonctions. Cela permet de structurer les programmes et d'organiser leur écriture : on parle de **conception modulaire.**

I Définition, paramètres et valeurs retournés

Supposons que l'on souhaite disposer d'une fonction qui renvoie la valeur d'un nombre s'il est positif et 0 sinon. On écrira :

```
1 def partie_positive(x):
2     """ Renvoie x s'il est positif et 0 sinon """
3
4     if x >= 0:
5         return x
6     else:
7         return 0
```

1

Après avoir exécuté le code correspondant, on dispose d'une nouvelle fonction que l'on peut **appeler** :

```
1 >>> partie_positive(-5)
2 0
3 >>> a = 2
4 >>> b = partie_positive(a + 2) + 3
5 >>> b
6 7
```

2

La **syntaxe générale** pour définir une fonction est :

```
1 def nomFonction(param1, param2, ...):
2     instruction
3     instruction          # bloc d'instructions
4     instruction
5     ...
```

3

Les paramètres utilisés pour définir la fonction sont appelés **paramètres formels**. Lors de l'appel de la fonction ils prennent une valeur qui s'appelle alors **argument ou paramètre effectif** : c'est cette **valeur** qui est transmise [1]. L'appel d'une fonction commence donc par l'évaluation des arguments [2].

1. Ceci est valable pour tous les types non mutables : les entiers, les flottants, les chaines de caractères, les tuples, les booléens. Ce n'est pas le cas pour les listes comme on le verra dans le chapitre qui leur est réservé.
2. En respectant l'ordre des arguments dans la définition de la fonction.

Ensuite le corps de la fonction, avec toutes ses instructions, est exécuté. Lorsque Python rencontre l'instruction **return** cela a simultanément deux conséquences :

* L'exécution du corps de la fonction est interrompue et on reprend la séquence d'instruction là où a eu lieu l'appel.
* L'objet qui figure derrière **return** donne sa valeur à la fonction dans l'expression où a eu lieu l'appel comme dans : `b = partie_positive(a + 2) + 3`. C'est la **valeur retournée** par la fonction.

Une fonction peut ne pas avoir d'argument (mais les parenthèses sont obligatoires) :

```
1  def affiche_trait():
2      """ Affiche un trait de séparation """
3
4      print('--------------------------------------------------')
5      return
6      print('code mort')
```

4

```
1  >>> affiche_trait
2  <function affiche_trait at 0x105a1bea0>
3  >>> affiche_trait()
4  --------------------------------------------------
```

5

Une fonction peut avoir plusieurs paramètres et renvoyer plusieurs valeurs sous forme de tuple :

```
1  def div_euclidienne(a, b):
2      """ Calcule le quotient et le reste de la division
3      euclidienne de a par b pour a >= 0 et b > 0 """
4
5      q = 0
6      r = a
7
8      while r >= b:
9          q = q + 1
10         r = r - b
11
12     return (q, r)
```

6

```
1  >>> div_euclidienne(45, 7)
2  (6, 3)
```

7

Si on ne s'intéresse qu'au reste (par exemple) il suffit de déconstruire le tuple :

```
1  >>> (q, r) = div_euclidienne(45, 7)
2  >>> r
3  3
```

8

Les fonctions sont des objets de Python qui ont leur classe :

```
1  >>> type(div_euclidienne)
2  <class 'function'>
```

9

Une fonction peut donc être un paramètre d'une autre fonction.

Il ne faut pas confondre **return** et **print**.

```
1 def norme(x, y):
2
3     print(sqrt(x**2 + y**2))
```
10

```
1 >>> norme(3, 4)
2 5.0
3 >>> l = norme(3, 4)
4 5.0
5 >>> print(l)
6 None
```
11

Lorsqu'une fonction ne renvoie rien (instruction **return** « vide » ou corps de la fonction exécuté sans en rencontrer) la valeur retournée est None qui est l'unique objet du type NoneType. En quelque sorte, Python dit explicitement qu'il n'y a rien plutôt que de ne rien dire.

Exercice 1
Comment faut-il modifier le code de la fonction norme ?

RETENIR :
- La fonction **print** ne renvoie pas de valeur, elle opère une action ; afficher dans la console.
- L'instruction **return** spécifie la valeur qui sera renvoyée par la fonction lors de son appel et termine l'exécution de la fonction. On peut récupérer la valeur pour en faire ce que l'on veut y compris l'afficher.

Exercice 2 Expliquer ce qui suit :

```
1 >>> L = [2, 3, 5, 7, 11]
2 >>> L = L.append(13)
3 >>> L
4 >>> print(L)
5 None
```
12

Exercice 3 Écrire une fonction `somme_images(f, n)` qui calcule $\sum_{k=0}^{n} f(k)$.

Les exercices qui suivent ont pour but de montrer comment coder des suites récurrentes.

Exercice 4

Ecrire une fonction `suite_rec(n, f, a)` qui prend en paramètres un entier `n`, un flottant `a` et une fonction `f` et renvoie le terne de rang `n` de la suite définie par :

$$\begin{cases} u_0 &= a \\ u_{n+1} &= f(u_n) \end{cases}$$

Entraînement 1

Ecrire une fonction `suite(n, f, a)` qui prend en paramètres un entier `n`, un flottant `a` et une fonction `f` et renvoie le terne de rang `n` de la suite définie par :

$$\begin{cases} u_0 &= a \\ u_{n+1} &= f(n, u_n) \end{cases}$$

Exercice 5

Écrire une fonction `suite(n)` qui prend en paramètres un entier `n` et renvoie le terne de rang `n` de la suite définie par :

$$\begin{cases} u_0 = 1,\ u_1 = 0 \\ u_{n+2} = n^2 u_{n+1} + 2u_n \end{cases}$$

Entraînement 2

Écrire une fonction qui prend en paramètre un entier n et renvoie le terme de rang u_n de la suite.

$$(u_n) : \begin{cases} u_0 = 1,\ u_1 = 2,\ u_2 = 3 \\ u_{n+3} = u_n u_{n+1} + n^2 u_{n+2} \end{cases}$$

Entraînement 3 Écrire une fonction qui prend en paramètres un entier `n` et deux flottants `x` et `y` et calcule les termes a_n et b_n des suites définies par :

$$\begin{cases} a_0 = x,\ b_0 = y, \\ a_{n+1} = \dfrac{a_n + b_n}{2} \\ b_{n+1} = \sqrt{a_n b_n} \end{cases}$$

Exercice 6

On exécute le code suivant :

```
def machin(x):
    print(2*x)
    return x+1
    print('fini !')
```

Prévoir la réponse de la console pour :

```
>>> machin(5)


>>> a = machin(5)

>>> a

```


II Variables locales et variables globales

Pour réaliser son action, une fonction a généralement besoin de variables. Les variables définies à l'intérieur du corps d'une fonction sont dites **locales** par opposition à celles qui font partie du programme principal qui sont dites **globales**.

```
def racine_entiere(n):
    """ renvoie le plus grand entier dont le carré est inférieur
    ou égal à n """
    r = 0
    while r**2 <= n:
        r = r + 1
    return r - 1
```

13

Les variables locales n'existent pas à l'extérieur du corps de la fonction :

```
>>> racine_entiere(26)
5
>>> r
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    r
NameError: name 'r' is not defined
>>> r = 2
>>> racine_entiere(26)
5
>>> r
2
>>>
```

14

Si une variable `r` existe avant l'appel de la fonction, l'affectation d'une valeur à une variable **locale** de même nom ne changera rien pour cette variable **globale**. On peut résumer en disant que le monde extérieur à une fonction ne voit rien de ce qui s'y passe.

Dans le sens inverse c'est un peu différent car le contenu des variables globales est accessible depuis l'intérieur de la fonction.

```
>>> def affiche_jour():
        print(jour)

>>> jour = 'lundi'
>>> affiche_jour()
lundi
```

15

Exercice 7 Prévoir le résultat :

```
>>> def affiche_double_n():
        n = 2*n
        print(n)

>>> n
5
>>> affiche_double_n()
```

16

Pour avoir accès en écriture aux variables globales à l'extérieur de la fonction il faut utiliser l'instruction **global**.

```
>>> def double_n():
        global n
        n = 2*n

>>> n = 5
>>> double_n()
>>> n
10
```

17

Exercice 8

Expliquer :

```
>>> truc
Traceback (most recent call last):
  File "<pyshell#73>", line 1, in <module>
    truc
NameError: name 'truc' is not defined
>>> def chose():
        global truc
        truc = 'machin'

>>> chose()
>>> truc
'machin'
```

18

Attention : L'utilisation de l'instruction **global** est fortement déconseillée car on peut modifier une variable globale sans en avoir conscience. Il faut au moins bien comprendre que l'on joue avec le feu ...

Exercice 9 On exécute le code de l'éditeur, prévoir le résultat dans la console :

```
A = 10
def f(x):
    y = 2*A
    return y - x
```

```
>>> A

>>> f(5)

>>> x

```

Exercice 10

Dans le code suivant, quel est le statut de chaque variable.

```
def f():
    global a
    a = a + 1
    c = 2 * a
    return a + b + c
```

Il est possible d'imbriquer des fonctions :

```
def est_divisible(a, b):
    def div_euclidienne(a, b):
        q = 0
        r = a
        while r >= b:
            q = q + 1
            r = r - b
        return q, r
    (q, r) = div_euclidienne(a, b)
    return r == 0
```

19

Dans cette situation, il y a une **hiérarchie des noms**. Chaque fonction a accès à son espace de noms et à ceux des fonctions situées au-dessus.

Comme on peut s'y attendre, la fonction `div_euclidienne` n'est pas accessible au niveau du programme principal.

```
>>> est_divisible(15, 3)
True
>>> div_euclidienne(15, 4)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    div_euclidienne(15, 4)
NameError: name 'div_euclidienne' is not defined
```

20

Le constructeur **lambda** permet de définir une fonction sans lui donner de nom.

```
>>> sommeImages(lambda x : x**2, 10)
385
```

21

Ne pas en abuser car il peut rendre un code difficile à lire.

Exercice 11 Que donne chacun des codes suivants :

```
n = 0
def g(x):
    global n
    n = n + 1
    return x + n
def somme(x, y):
    return x + y
print(somme(n, g(1)))
```

```
n = 0
def g(x):
    global n
    n = n + 1
    return x + n
def somme(x, y):
    return x + y
print(somme(g(1), n))
```

III Effet de bord, lecture d'énoncé et bonnes pratiques

Généralement, une fonction renvoie une valeur après avoir reçu ses paramètres de l'extérieur du code. Mais ce n'est pas toujours le cas. On appelle souvent **procédure** une fonction qui ne renvoie rien. Une telle fonction peut être très utile en réalisant une action (par exemple modifier une liste), c'est un **effet de bord**. Une fonction peut également faire les deux. Il est essentiel de comprendre la différence.

De nombreuses questions des sujets de concours vous demanderont d'écrire des fonctions, il s'agit de bien comprendre ce que l'on attend de vous.

Un extrait du sujet de centrale 2016 :

Écrire en Python une fonction `recuit(regulation)` *qui modifie la liste* `regulation` *passée en paramètre en appliquant l'algorithme du recuit simulé.*

Ici c'est clairement un **effet de bord** qui est **attendu.** Ce sera le cas de toutes les questions formulées de cette manière.

Plus généralement on aura, par exemple :

Écrire en Python une fonction `nb_conflits()` *sans paramètre qui renvoie le nombre de conflits potentiels, c'est-à-dire le nombre d'arêtes de valuation non nulle du graphe.*

ou encore :

Écrire en Python une fonction `cout_regulation(regulation)` *qui prend en paramètre une liste représentant une régulation et qui renvoie le coût de celle-ci.*

Dans ces deux derniers cas votre fonction ne **doit pas avoir d'effet de bord**. Ce serait une erreur : « effet de bord non désiré ».

Il faut toujours essayer de rédiger son code de la manière la plus claire et la plus simple. La première idée n'est pas toujours la plus efficace ni la plus facile à coder. L'utilisation des docstrings (on a accès à cette docstring par la fonction **help**) et des commentaires peut permettre de lever les ambiguïtés et difficultés éventuelles. Se relire sans aucune complaisance est un bon début.