Les fonctions en Python

Programme officiel :

- Prototyper une fonction
- Décrire les préconditions sur les arguments
- Décrire des postconditions sur les résultats

Introduction

Vous commencez à écrire vos premiers programmes, ces derniers deviennent de plus en plus complexes, des parties sont redondantes (il est alors tentant de faire un copier-coller de votre code).

Pour obtenir des programmes plus courts et plus lisibles, il est d'usage de définir des fonctions qui vont de décomposer un programme complexe en une série de sous-programme plus simples.

Une fonction est un bloc d'instructions (une sorte de sous-programme) que l'on peut appeler au tant de fois que l'on veut et à tout moment d'un programme. La notion de fonction en informatique est comparable à la notion de fonction en mathématiques.

En résumé : Les fonctions ont plusieurs intérêts, notamment :

- La réutilisation du code : éviter de répéter les mêmes séries d'instructions à plusieurs endroits d'un programme ;
- La modularité : découper une tâche complexe en plusieurs sous-tâches plus simples.
- S'il y a un bug ou une modification à apporter, il suffira de corriger et/ou modifier la fonction plutôt que de rechercher dans l'ensemble du programme.

Implémentation d'une fonction

Les fonctions sont des bouts de code d'un programme qui sont exécutés à chaque fois qu'ils sont appelés.

Pour définir une fonction en python, il faut lui attribuer un nom clair et explicite.

Par exemple, `affiche_menu` est un nom clair décrivant le rôle de cette fonction et facile à retenir comparer à un nom plus court donné à la va-vite tel que `f1`.

Par convention, les noms de fonction sont écrits en minuscules et la séparation entre mots est délimité par un `_`.

Il faut également noter que rien n'empêche une fonction d'en appeler une autre. Par exemple `affiche_menu` pourrait d'abord appeler une fonction `efface_ecran`.

Une fonction peut même s'appeler elle-même : c'est la `récursivité`.

Pour implémenter une fonction en python, on utilise la structure suivante :

```
def nom_fonction():
    """                              # docstring entouré de trois
    Documentation de la fonction     # guillemets (ou apostrophes)
    """
    les instructions        # attention à l'indentation
    return resultat         # la fonction retourne le contenu de la
                            # variable résultat
```

`def` est le mot clé indiquant à Python que l'on est en train de définir une fonction. `nom_fonction` est le nom de la fonction. C'est celui que nous utiliserons pour l'appeler. Viens ensuite le code de la fonction qui sera exécuté à chaque appel.

Notez bien l'indentation du code dans la fonction. La réduction d'un niveau d'indentation délimite la fin de la fonction. C'est le même principe que pour les conditions ou les boucles.

```
# exemple de fonction
def ma_fonction():
    '''cette fonction affiche Bonjour'''
    print("Bonjour !")
    return             # cette fonction ne retourne rien ('None')
                       # l'instruction return est ici facultative
```

Une fois la fonction définie, on peut l'appeler :

```
>>> ma_fonction()    # ne pas oublier les parenthèses ()
Bonjour !
```

La documentation de la fonction (docstrings) est très importante, nous y reviendrons un peu plus loin.

Remarque : une fonction qui ne renvoie pas de résultat (comme ci-dessus) est appelée une `procédure`, il n'y a pas de différence en Python entre la notion de `fonction` et la notion de `procédure` mais les objectifs sont différents :

- Une procédure réalise une action : écrire dans un fichier, afficher quelque chose, faire un dessin avec Turtle...
- Une fonction a pour objectif de calculer et de mettre à disposition une valeur

---

Renvoi d'une valeur par une fonction

Créons une fonction qui une fois appelée renvoie un chiffre aléatoire compris entre 1 et 6 à l'image d'un dé à six faces.

Pour cela, on utilisera la fonction `randint()` du module `random` dont la documentation est disponible ici, il faudra donc importer le module `random` au début du programme.

```
import random        # import du module random
def lancer_de():
    valeur = random.randint(1,6)    # génération d'un nombre entier
    return valeur                   # renvoi de la valeur
```

On peut alors tester la fonction `lancer_de` :

```
>>> lancer_de()
3
>>> lancer_de()
1
```

Fonction utilisant un paramètre

Certains dés ont la forme d'un polyèdre autre que le cube. Jadis peu employés dans le jeu, ils sont devenus plus populaires depuis les années 1950, particulièrement après l'introduction des wargames et autres jeux de rôle. Ces dés comportent des nombres de faces très variables, pour les plus courant : 8, 10, 12, 20.

Créons maintenant une fonction s'inspirant de la fonction `lancer_de` mais avec la possibilité de passer en paramètre le nombre de faces du dé.

```
import random             # import du module random
def lancer_de(nb_faces):
    valeur = random.randint(1,nb_faces) # génération d'un nombre
    return valeur                       # renvoi de la valeur
```

On peut alors tester la fonction `lancer_de` :

```
>>> lancer_de(12)
9
>>> lancer_de(12)
11
```

Il faut par contre ne pas oublier d'indiquer le paramètre `nb_faces` sinon on obtient un message d'erreur indiquant l'oubli d'un argument :

```
>>> lancer_de()
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: lancer_de() missing 1 required positional argument: 'nb_faces'
```

Pour éviter ce type d'erreur, on peut indiquer un paramètre qui sera pris par défaut si aucune valeur n'est spécifiée par l'utilisateur de la fonction : on parle de `paramètre optionnel`.

Dans notre fonction `lancer_de` si aucun nombre de faces n'est donné, on considérera que notre dé comporte six faces.

Pour cela, on écrira :

```
import random                          # import du module random
def lancer_de(nb_faces = 6):
    valeur = random.randint(1,nb_faces) # génération d'un nombre
    return valeur                       # renvoi de la valeur
```

On obtient alors :

```
>>> lancer_de()
4
>>> lancer_de(20)
14
```

Fonction utilisant plusieurs paramètres

Vous pouvez avoir plusieurs paramètres pour une seule fonction. Il vous suffit de les séparer par des virgules.

```
def addition(a, b):
    return a + b

>>> addition(10, 5)
15
```

Remarque : le calcul de a + b se fait directement dans l'instruction `return` ce qui raccourci le code.

Portée des variables

La portée d'une variable est l'endroit du programme à partir duquel on peut accéder à la variable.

On distinguera les `variables locales` (à la fonction) et les `variables globales` (à tout le programme).

Soit le script ci-dessous pour mettre en évidence la différence :

```
a = 10                  # variable globale au programme

def mafonction():
    a = 20              # variable locale à la fonction
    print(a)
    return
```

On observe :

```
>>> print(a)        # nous sommes dans l'espace global du programme
10
>>> mafonction()    # nous sommes dans l'espace de la fonction
20
>>> print(a)        # nous sommes de retour dans l'espace global
10
```

Nous avons deux variables différentes qui portent le même nom `a`

Une variable `a` de valeur 20 est créée dans la fonction : c'est une `variable locale` de la fonction, elle est détruite dès que l'on sort de la fonction.

Remarque : On peut observer cela en utilisant le mode debug de Thonny ou Python tutor !

Dans la plupart des langages, une variable définie dans une fonction est utilisable uniquement dans cette fonction. En dehors de cette fonction, la variable n'existe pas !

L'instruction `global` permet de rendre une variable globale :

```
a = 10              # variable globale

def mafonction():
    global a        # la variable est maintenant globale
    a = 20
    print(a)
    return
```

On observe :

```
>>> print(a)         # nous sommes dans l'espace globale
10
>>> mafonction()     # la fonction modifie la variable a qui est
20                   # ici globale
>>> print(a)         # retour dans l'espace globale avec la
20                   # variable qui a été modifiée
```

Dans le programme ci-dessus le `global a` signifie "il faut utiliser la variable a" qui a été définie en dehors de la fonction.

Remarques :

- Il est préférable d'éviter l'utilisation de l'instruction `global` car c'est une source d'erreurs (on peut ainsi modifier le contenu d'une variable globale en croyant agir sur une variable locale).
- Il ne faut donc jamais affecter dans un bloc de code local une variable de même nom qu'une variable globale.
- Si on utilise une variable affectée au niveau globale dans une fonction, la fonction pourra y accéder mais pas la modifier.

Docstrings et spécification d'une fonction

Lorsqu'on définit des fonctions, c'est évidemment dans le but de les utiliser. Il est donc très important de les documenter, c'est-à-dire d'ajouter un texte expliquant ce qu'elles font, et comment les utiliser.

On pourrait par exemple écrire :

```
def isdivisor(d, n):
    ''' fonction qui permet de tester si le nombre entier d est un
    diviseur du nombre entier n '''
    return n % d == 0
```

Ce texte appelé `docstrings` décrit ce que fait la fonction, ainsi que les paramètres qu'il faut lui fournir. Il permet d'utiliser la fonction comme il faut.

Dans le `shell` , on peut obtenir les `docstrings` en tapant :

```
>>> help(isdivisor)
Help on function isdivisor in module __main__:

isdivisor(d, n)
    fonction qui permet de tester si le nombre entier d est un
    diviseur du nombre entier n
```

Remarque : Les docstrings ne sont pas des commentaires (qui commencent avec des #) car les commentaires sont ignorés par l'interpréteur ce qui n'est pas le cas des docstrings.

Mais on peut décrire une fonction de manière plus systématique en fournissant sa `spécification`. Pour cela, on doit décrire deux choses :

- Les `préconditions` d'une fonction sont toutes les conditions qui doivent être satisfaites avant de pouvoir appeler la fonction, que ce soit sur des variables globales ou sur ses paramètres ;
- Les `postconditions` d'une fonction sont toutes les conditions qui seront satisfaites après appel de la fonction, si les `préconditions` étaient satisfaites, que ce soit sur des variables globales ou sur l'éventuelle valeur renvoyée.

On pourrait spécifier la fonction `isdivisor` en définissant ses préconditions et postconditions :

```
def isDivisor(d, n):
    '''' Test de la divisibilité d'un nombre par un autre
    Préconditions : d et n sont deux entiers positifs
                    d doit être différent de 0
    Postconditions : la valeur renvoyée est un booléen qui vaut True si d
                     divise n, et False sinon ''''
    return n % d == 0
```

Pour appeler la fonction, il faut donc lui fournir deux nombres entiers positifs en paramètres, et s'assurer que d soit différent de zéro. Dans ce cas, après avoir appelé la fonction, la valeur qu'elle aura renvoyée contiendra True si d est un diviseur de n et False sinon.

La `spécification` d'une fonction contient donc toute la documentation nécessaire pour l'utiliser correctement. S'il ne faut pas devoir lire le corps de la fonction pour comprendre ce qu'elle fait et comment l'utiliser, c'est que la `spécification` est bien écrite.

Afin d'harmoniser, la manière de spécifier une fonction, nous utiliserons l'écriture suivante :

```
def isdivisor(d, n):
''' Test de la divisibilité d'un nombre par un autre
:param d: le nombre a tester
:param n: le diviseur
:type a: int
:type b: int
:return: True si d divise n, False sinon
:rtype: bool
:CU: d et n doivent être positifs, et d ne doit pas être nul
:effet de bord: aucun
'''
return n % d == 0
```

Assertions sur les fonctions

Lorsqu'on définit une nouvelle fonction, et qu'on la spécifie, il faut minimiser le nombre de `préconditions` si on veut la rendre robuste, c'est-à-dire résistante à des mauvaises utilisations.

On peut vouloir vérifier que des conditions qui sont censées être satisfaites le sont effectivement, à l'aide du `mécanisme d'assertion` proposé par Python (on parle de programmation défensive).

Par exemple :

```
def pourcentage(valeur, total):
    """ fonction qui permet de calculer le pourcentage d'une valeur par
    rapport à un total.
    :param valeur: valeur à calculer
    :param total: total par rapport auquel calculer le pourcentage
    :type valeur: int
    :type total:
    :return: le pourcentage
    :rtype: float
    :CU: la valeur et le total doivent être positifs, la valeur doit être
    inférieur au total
    :effet de bord: aucun
    """
    assert total > 0, 'total doit être strictement positif'
    assert 0 <= valeur, 'valeur doit être positif'
    assert valeur <= total, 'valeur doit être inférieur à total'

    return valeur / total * 100
```

Trois instructions `assert` ont été utilisées pour vérifier les préconditions. Une telle instruction se compose d'une condition (une expression booléenne) éventuellement suivie d'une virgule et d'une phrase en langue naturelle, sous forme d'une chaine de caractères.

L'instruction `assert` teste si la condition est satisfaite. Si c'est le cas, elle ne fait rien et sinon elle arrête immédiatement l'exécution du programme en affichant la phrase qui lui est éventuellement associée.

```
>>> pourcentage(56,120)
46.666666666666664

>>> pourcentage(0,120)
0.0

>>> pourcentage(10,0)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "C:\Users\Frédéric\Desktop\essai python\test4.py", line 2, in pourcentage
    assert total > 0, 'total doit être strictement positif'
AssertionError: total doit être strictement positif

>>> pourcentage(10,5)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "C:\Users\Frédéric\Desktop\essai python\test4.py", line 4, in pourcentage
    assert valeur <= total, 'valeur doit être inférieur à total'
AssertionError: valeur doit être inférieur à total

>>> pourcentage(-10,50)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "C:\Users\Frédéric\Desktop\essai python\test4.py", line 3, in pourcentage
    assert 0 <= valeur, 'valeur doit être positif'
AssertionError: valeur doit être positif
```

Le mécanisme d'assertion est là pour empêcher des erreurs qui ne devraient pas se produire, en arrêtant prématurément le programme, avant d'exécuter le code qui aurait produit une erreur. Si une telle erreur survient, c'est que le programme doit être modifié pour qu'elle n'arrive plus.

On peut aussi utiliser l'instruction `assert` pour tester une fonction, ainsi si on ajoute le code ci-dessous (en dehors de la fonction), le programme s'interrompra si les tests ne sont pas validés.

```
assert pourcentage(5, 100) == 5, 'test avec 5% non valide'
assert pourcentage(0, 100) == 0, 'test avec 0 comme valeur non valide'
```

Enfin, il faut savoir que le mécanisme d'assertion est une aide au développeur, et ne doit en aucun cas faire partie du code fonctionnel d'un programme. Il faudra pour cela gérer les erreurs en traitant les exceptions avec d'autres outils.

Doctests d'une fonction

Mettre au point des `doctests` consiste à ajouter des exemples de résultats donnés par une fonction dans la chaîne de documentation (`docstrings`) . Ces exemples peuvent alors être testés à l'aide d'un module de Python nommé `doctest` .

Par exemple (volontairement basique) :

```
def addition(a, b):
    """Une fonction qui additionne deux entiers
    :param a: premier entier à additionner
    :param b: deuxième entier à additionner
    :type a: int
    :type b: int
    :return: l'addition de a et de b
    :rtype: int
    :Example:
    >>> addition(3, 5)
    8
    >>> addition(2, 8)
    10
    >>> addition(9, 15)
    24
    """
    return a + b

import doctest
doctest.testmod(verbose = True)
```

Quand on lance le programme ci-dessus, rien ne s'affiche (et c'est normal)

Si on modifie les `doctests` en ajoutant (volontairement) un test eronné :

```
def addition(a, b):
    """Une fonction qui additionne deux entiers
    :param a: premier entier à additionner
    :param b: deuxième entier à additionner
    :type a: int
    :type b: int
    :return: l'addition de a et de b
    :rtype: int
    :Example:
    >>> addition(3, 5)
    8
    >>> addition(2, 8)
    10
    >>> addition(9, 15)
    24
    >>> addition(5, 5)
    11
    """
    return a + b

import doctest
doctest.testmod()


# on obtient
```

Ce message montre qu'un des tests a échoué, le résultat attendu était 11 et la fonction a renvoyé 10.

En résumé, s'il les tests se passent sans erreur : rien ne s'affiche par contre un message d'erreur apparait dans le cas contraire.

Remarque : On peut écrire `doctest.testmod(verbose = True)` si on veut afficher le résultat des tests même s'il n'y a d'erreur.

La qualité et le nombre de tests sont importants dans la mise au point d'un programme, néanmoins le succès d'un jeu de tests ne garantit pas la correction d'un programme !