TP Python 2 - Fonctions

Définition

Une **fonction** est un ensemble d'instructions qui accomplit une tâche donnée et auquel on donne un nom. Les **paramètres** d'une fonction sont les variables avec lesquelles on la définit. Quand on appelle la fonction, on lui donne les valeurs (appelées **arguments**) que les paramètres doivent prendre.

En Python on définit les fonctions avec `def`. L'indentation marque le bloc d'instructions qui définit la fonction. L'instruction `return` permet de renvoyer un résultat.

On écrira les fonctions dans la zone de script et on les testera dans la console. Définissons par exemple la fonction inverse :

```
def inverse(x):   # x est le paramètre de la fonction
    return 1/x    # attention à l'indentation !
```

Ne pas oublier d'exécuter le programme (qui ne renvoie rien). On peut alors appliquer la fonction à différentes valeurs :

```
>>> inverse(5)        # on appelle la fonction avec 5 comme argument
0.2
>>> inverse(3)
0.3333333333333333
```

La fonction suivante a pour paramètre une chaîne de caractères :

```
def saluer(nom):
    print("Bonjour " + nom)
    print("Comment allez-vous ?")
```

```
>>> saluer("Léon")
Bonjour Léon
Comment allez-vous ?
```

Exercice 1 Écrire une fonction `conversion` qui convertit les euros en dollars (on prendra 1,16 dollars pour 1 euro).

```
>>> conversion(10)
11.6
```

Une fonction peut avoir plusieurs paramètres (ou aucun) :

```
def somme_de_carres(a, b):  # cette fonction a deux paramètres
    return a**2 + b**2
```

```
>>> somme_de_carres(1, 2)
5
```

```
def dire_bonjour():         # une fonction sans paramètres
    print("Bonjour")
```

```
>>> dire_bonjour()          # il faut quand même mettre les parenthèses
Bonjour
```

Exercice 2

1) Écrire une fonction `min_2` qui, recevant deux nombres, renvoie le plus petit.

2) Écrire une fonction `min_3` qui, recevant trois nombres, renvoie le plus petit.

```
>>> min_2(4, 2)
2
>>> min_3(2, 1, 3)   # à tester avec différentes valeurs
1
```

Différences entre `print` et `return`

L'instruction `return` *renvoie* un résultat (qui pourra être réutilisé ensuite) et arrête l'exécution de la fonction, alors que la fonction `print` ne fait qu'*afficher* un résultat et n'arrête pas l'exécution. Noter que les parenthèses ne sont pas nécessaires après `return`.

```
def f(x):
    return x**2 + 1
    print("abc")
```

```
>>> f(3)
10                  # le print("abc") n'est pas exécuté
>>> f(3) + 5
15                  # on peut utiliser le résultat dans un autre calcul
```

```
def g(x):
    print(x**2 + 1)
```

```
>>> g(3)
10
>>> g(3) + 5
10
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

La fonction `g` affiche `10` mais elle ne renvoie rien (plus précisément elle renvoie `None`).

Exemples

La fonction suivante reçoit un entier naturel non nul n et renvoie $\sum_{k=1}^{n} \frac{1}{k}$:

```
def somme_harmonique(n):
    s = 0
    for k in range(1, n+1):
        s = s + 1/k
    return s
```

```
>>> somme_harmonique(10)
2.9289682539682538
```

Exercice 3

1) Écrire une fonction `factorielle` qui reçoit un entier naturel `n` et renvoie `n!` (par exemple `factorielle(5)` doit renvoyer `120`).

2) Écrire une fonction `seuil` qui reçoit un entier `M` et renvoie le plus petit entier naturel `n` tel que `n! > M` (par exemple `seuil(100)` doit renvoyer `5`).

La fonction suivante reçoit un entier naturel et renvoie un booléen indiquant si cet entier est premier ou non :

```
def est_premier(n):
    if n < 2:              # 0 et 1 ne sont pas premiers
        return False
    for k in range(2, n):
        if n % k == 0:     # i.e. si n est divisible par k
            return False
    return True
```

```
>>> est_premier(5)
True
>>> est_premier(6)
False
>>> for n in range(10):
...     if est_premier(n):
...         print(n)
2
3
5
7
```

Au cours de l'exécution d'une fonction, on peut utiliser d'autres fonctions. Exemple :

```
def nombres_premiers(n):
    for k in range(n+1):
        if est_premier(k):
            print(k, "est premier")
        else:
            print(k, "n'est pas premier")
```

```
>>> nombres_premiers(5)
0 n'est pas premier
1 n'est pas premier
2 est premier
3 est premier
4 n'est pas premier
5 est premier
```

Pour définir une fonction on utilisera ainsi souvent des fonctions auxiliaires.

Variables locales et globales

Les variables définies à l'intérieur d'une fonction sont des **variables locales**, elles n'existent que dans une partie de la mémoire réservée à la fonction quand on l'exécute.

```
def f():
    a = 32
    print(a)
```

```
>>> a
NameError: name 'a' is not defined
>>> f()
32
>>> a
NameError: name 'a' is not defined
```

Les variables définies en-dehors des fonctions sont appelées **variables globales**, elles ne sont pas modifiées quand on exécute une fonction.

```
a = 1
b = 2
def f():
    a = 3
    print(a, b)
```

```
>>> f()
3 2
>>> a
1          # a n'a pas été modifiée
```

Dans cet exemple, il n'y a aucun lien entre le `a` défini au début et le `a` défini dans le corps de la fonction car ils sont créés dans des espaces de noms (*namespaces* en anglais) différents.

Si on veut créer ou modifier des variables globales dans une fonction, il faut les déclarer comme telles au début de la fonction.

```
a = 1
def f():
    global a   # on déclare a comme variable globale
    a = 3
    print(a)
```

```
>>> f()
3
>>> a
3          # a a été modifiée
```

Documenter une fonction

En Python, on peut documenter une fonction à l'aide d'une *docstring* : c'est une chaîne de caractères délimitée par des triples guillemets que l'on place sous l'en-tête de la fonction. La fonction `help` permet d'afficher la docstring. Elle apparaît également quand on tape le nom de la fonction dans la console.

```
def somme(a, b):
    '''Renvoie la somme de a et b.'''
    return a + b
```

```
>>> help(somme)
Help on function somme in module __main__:
somme(a, b)
    Renvoie la somme de a et b.
```

Exercices

Un bon programme ne doit pas seulement fonctionner correctement, il doit aussi pouvoir être lu ou relu facilement. Prenez dès maintenant de bonnes habitudes de programmation :

– Précisez les spécifications de vos fonctions (ce qu'elles reçoivent, ce qu'elles renvoient) à l'aide d'une docstring ou d'un commentaire.

– Commentez les passages délicats de vos programmes en évitant la paraphrase.

– Après avoir écrit une fonction, testez-la sur plusieurs exemples (pas seulement celui donné dans l'énoncé) qui recouvrent les différents cas possibles. Ainsi, il convient de tester la fonction `est_bissextile` ci-dessous avec une année divisible par 400, avec une année divisible par 100 mais pas par 400, avec une année divisible par 4 mais pas par 100 et avec une année non divisible par 4.

Exercice 4 Les années divisibles par 4 sont bissextiles, sauf si elles sont divisibles par 100 (mais les années divisibles par 400 sont bissextiles). Par exemple, 2012 et 2000 étaient bissextiles, mais 1900 ne l'était pas. Écrire une fonction `est_bissextile` qui, recevant une année, renvoie `True` si elle est bissextile et `False` sinon.

```
>>> est_bissextile(2019)
False
```

Exercice 5 Écrire une fonction `cubes` qui, recevant un entier naturel n, affiche les cubes des entiers compris entre 0 et n.

```
>>> cubes(3)
0
1
8
27
```

Exercice 6 Écrire une fonction `equation_second_degre` qui, recevant trois réels a, b et c, affiche le nombre de solutions réelles de l'équation $ax^2 + bx + c = 0$ et leurs valeurs.

```
>>> equation_second_degre(2, 3, 1)
Deux solutions réelles :  -1.0 -0.5
>>> equation_second_degre(1, 2, 1)
Une solution réelle double : -1.0
>>> equation_second_degre(1, 1, 1)
Pas de solutions réelles
```

Le résultat suivant est-il correct ? Expliquer et proposer une solution.

```
>>> equation_second_degre(0.1, 0.6, 0.9)
Pas de solutions réelles
```

Exercice 7

1) Écrire une fonction `jhms` qui, recevant un nombre entier de secondes, le convertit en jours, heures, minutes, secondes.

2) Écrire une fonction `secondes` qui, recevant un nombre entier de jours, heures, minutes, secondes, renvoie le nombre de secondes correspondant.

```
>>> jhms(100000)
Jours : 1
Heures : 3
Minutes : 46
Secondes : 40
>>> secondes(1, 3, 46, 40)
100000
```

Exercice 8 *(Suite de Syracuse)* On considère l'algorithme suivant. Un entier naturel n non nul est donné. S'il est pair, on le divise par 2. S'il est impair, on le multiplie par 3 et on ajoute 1. Puis on recommence : si le nombre obtenu est pair, on le divise par 2, et s'il est impair on le multiplie par 3 et on ajoute 1. Et on continue...

1) Appliquer cet algorithme (à la main) à $n = 1, 2, 3, 4, 5, 6, 7$. Que peut-on conjecturer ?

2) Écrire une fonction qui, recevant un entier n, lui applique l'algorithme et affiche la suite des nombres obtenus jusqu'à ce qu'on arrive à 1.

3) Pour quel entier $n \leqslant 100$ le nombre d'itérations effectuées est-il maximal ?

Pour les exercices suivants, on pourra utiliser la fonction `est_premier` de la page précédente.

Exercice 9 Trouver 10 nombres non premiers consécutifs.

Exercice 10 La conjecture de Goldbach affirme que tout entier naturel pair strictement supérieur à 2 peut s'écrire comme somme de deux nombres premiers.

1) Écrire une fonction `goldbach` qui, recevant un entier naturel n non nul, affiche, pour chaque entier pair compris entre 4 et n, une décomposition de cet entier en somme de deux nombres premiers.

```
>>> goldbach(10)
4 = 2 + 2
6 = 3 + 3
8 = 3 + 5
10 = 3 + 7
```

2) Vérifier la conjecture pour les entiers inférieurs ou égaux à 10000.

Exercice 11 Deux nombres premiers sont **jumeaux** si leur différence est égale à 2. Par exemple, 5 et 7 sont jumeaux, 11 et 13 aussi. Trouver tous les couples de nombres premiers jumeaux compris entre 1 et 1000.