Python: fonctions, arguments, return, docstring, var local/global

- Auteur: Valentin Piquerez
- Classe: 3MOCINFO
- Date: 31 Mai 2019

Fonctions

Une **fonction** est une séquence d'instructions nommée, qui est **définie avant son utilisation**, et qui peut être appelée multiple fois. On injecte des **arguments** dans une fonction et elle peut retourner des valeurs. Les fonctions sont **executées** dans leur **ordre d'apparition dans le code**.

Fonctions prédefinies

Sur python, il y a des fonctions de base comme la fonction `type` qui retourne le type de ce qu'on lui soumet. Voici certains exemples de ce que la fonction peut retourner:

- `int` (nombre entier)
- `float` (nombre à virgule)
- `str` (chaîne de caractères)

```
In [1]: type(42), type(42.42), type('quarante-deux')
```
```
Out[1]: (int, float, str)
```

Il est aussi possible de changer le type avec les **fonctions de conversions de type**. Toutefois, cela ne fonctione que si la conversion est possible.

```
In [2]: int(42.42), int('25')
```
```
Out[2]: (42, 25)
```
```
In [3]: float(2), float('42'), float('25.25')
```
```
Out[3]: (2.0, 42.0, 25.25)
```
```
In [4]: str(2), str(42.25)
```
```
Out[4]: ('2', '42.25')
```

Il est impossible par exemple de transformer 'quarante-deux' en 42 avec la fonction `int`.

Fonctions mathématiques

De base, seul 7 opérations mathématiques sont disponibles.

- L'addition (+)
- La Soustraction (-)
- La multiplication (*)
- La puissance (**)
- La division (/)
- La division entière (//)
- Le modulo (%)

Il est tout de même possible d'avoir plus d'opérations en important le module **math** grâce à cet commande:

```
In [5]: import math
```

Avec ce module, on peut faire des **racines**, des fonctions **trigonométriques** et plus encore. La liste complète des opérations figure ici. Pour l'utiliser, il faut l'importer puis écrire `math.` suivi de l'opération.

Voici quelques exemples:

```
In [6]: math.sqrt(25)
```
```
Out[6]: 5.0
```
```
In [7]: math.log10(25)
```
```
Out[7]: 1.3979400086720377
```
```
In [8]: math.sin(1)
```
```
Out[8]: 0.8414709848078965
```

Il est biensur possible de combiner des opérations.

```
In [9]: math.sqrt(math.log10(math.pi))
```
```
Out[9]: 0.7050885566325225
```

Créer une fonction

Il est aussi possible de **créer** sa propre fonction, ce qui donne une **infinité de possibilitées**. Pour ce faire, il faut commencer par définir sa fonction tel que ci-dessous avec le mot-clé `def`.

```
In [10]: def nom_de_la_fonction(arguments):
             pass
```

Ensuite, on peut créer une ou plusieurs instructions qui seront executées lorsque la fonction sera appellée. Il faut tout de même respecter une syntaxe où chaque ligne doit être décalée de 4 espaces par rapport au `def`, puis il faut écrire `return` à la fin, suivi de ce que l'on souhaite que la fonction retourne. Ces fonctions sont appelées **productives** car on obtient un *output*.

```
In [11]: def aire_carré(x):
             ###x is one side of the square###
             Aire_c = x**2
             return Aire_c

         def aire_rectangle(x, y):
             ###x and y are both sides of the rectangle###
             Aire_r = x*y
             return Aire_r
```

Pour *appeler la fonction* par la suite, il faut **écrire son nom** suivi des **arguments** dont elle a besoin entre **parenthèse**, séparés par des *virgules* si il y en a *plusieurs*.

```
In [12]: aire_carré(5), aire_rectangle(2, 3)
```
```
Out[12]: (25, 6)
```

Il est aussi possible d'utiliser des **variables** comme arguments.

```
In [13]: côté_carré = 5
         aire_carré(côté_carré)
```
```
Out[13]: 25
```

Pour **sortir** d'une fonction *sans que rien ne se passe*, il faut utiliser le mot-clé `pass`.

```
In [14]: def rien_ne_se_passe_si_plus_grand_que_2(x):
             if x > 2:
                 pass
             else:
                 return x

         rien_ne_se_passe_si_plus_grand_que_2(5)
         rien_ne_se_passe_si_plus_grand_que_2(1)
```
```
Out[14]: 1
```

Un autre type de fonction, appelé **fonction vide**, retourne comme valeur `None`. Par contre, ces fonctions printent quelque chose dans la **console**. Ces fonctions n'ont pas un `return` mais un `print()` à la fin.

```
In [15]: def addition(a, b):
             x = a+b
             print(x)

         addition(25, 42)
```
```
67
```
```
In [16]: print(addition(25, 42))
```
```
67
None
```

La fonction `None` appartient à la classe **NoneType**, qu'on n'a pas vu précédement.

```
In [17]: type(None)
```
```
Out[17]: NoneType
```

Arguments

Lorsqu'on appelle une fonction, on doit lui fournir des **arguments**. Ces *arguments* sont ensuite utilisés comme **paramètres variables** dans la fonction.

```
In [18]: def incrémenter(argument):
             argument += 1
             return argument

         x=2
         incrémenter(x)
```
```
Out[18]: 3
```

Dans ce cas, la *valeur* de *x* est **changée en paramètre** et donc varie *seulement dans la fonction*, tandis qu'elle *reste la même hors de la fonction*.

```
In [19]: print(x)
```
```
2
```

On peut attribuer à une **fonction** des arguments par **défaut** en mettant un `=` suivi de la valeur par défaut après un argument quand on le définit.

```
In [20]: import math
         def aire_triangle_equilateral(côté = 1):
             A = (math.sqrt(3)/4) * côté**2
             return A

         aire_triangle_equilateral(), aire_triangle_equilateral(5)
```
```
Out[20]: (0.4330127018922193, 10.825317547305483)
```

Return

Le mot-clé `return` a déjà été utilisé plus haut. Il se met dans les **fonctions** et permet de **sortir** de la fonction en donnant un **output**.

```
In [21]: def plus_deux(x):
             xbz = x+2
             return x

         plus_deux(4)
```
```
Out[21]: 4
```

Dès qu'on atteint un `return`, on sort de la fonction. Même si il y avais encore du code à executer.

```
In [22]: def plus_petit(x, y):
             if x < y:
                 return ('plus petit')
             return ('pas plus petit')

         plus_petit(2, 3), plus_petit(3, 2)
```
```
Out[22]: ('plus petit', 'pas plus petit')
```

Il est aussi possible de retourner un **tuple**. Pour cela, il faut mettre des *parenthèses avec* les éléments de la liste après le `return`.

```
In [23]: def trois_multiples(n):
             n1 = n*1
             n2 = n*2
             n3 = n*3
             return(n1, n2, n3)

         multiples_de_5 = trois_multiples(5)
         trois_multiples(5)
```
```
Out[23]: (5, 10, 15)
```

Sachant qu'on a une liste, on peut en extraire un élément.

```
In [24]: multiples_de_5[2]
```
```
Out[24]: 15
```

Docstrings

Les docstrings sont les **commentaires** apporté au code afin de mieux le comprendre et le lire. Les docstrings n'ont **aucun impact sur le code**, ils sont comme invisible. Généralement, on écrit les commentaires en **anglais** car c'est la langue de base de l'informatique.

Les docstrings sont placés à des endroits précis du code pour expliquer les parties concernées. Souvent, on les mets en **début de fonction** afin d'expliquer ce que cette dernière fait. Ils sont écrits entre `"""`.

```
In [25]: def function():
             """this fuction does ..."""
             pass
```

Si on veut rédiger un docstring de **plus d'une ligne**, il faut que le `"""` soit *après la dernière ligne de commentaire*.

```
In [26]: def function():
             """this function does...
             but it also does ...
             """
             pass
```

Ci-dessus, la convention PEP 257 est appliquée.

Variable locale/globale

Variable locale

Les variables local sont des variables **appartenant à une fonction**. On ne peut y **accéder** que si on est **dans la fonction**.

```
In [27]: def factoriel1(x):
             """Return the factorial of a number."""
             y = 1
             for i in range(x):
                 y = y*(i+1)
             return y

         factoriel1(5)
```
```
Out[27]: 120
```

Dans ce cas là, `y` **est une variables locale** car on ne peut y *accéder que dans la fonction*. Si on mettait un `print(y)` en dehors de la fonction, on aurais un *message d'erreur*.

Variable globale

Une **variable globale** est une variable **accessible dans les fonctions, mais aussi en dehors d'elles**.

```
In [28]: import math
         pi = math.pi

         def périmètre_cercle(rayon):
             péri = rayon * 2 * pi
             return péri

         périmètre_cercle(5), pi
```
```
Out[28]: (31.41592653589793, 3.141592653589793)
```

Dans cet exemple, on a assigné une valeur à `pi` en dehors de la fonction et on l'a appelé dans la fonctions. Si on veut **changer la valeur** d'une variable globale **dans une fonction**, il faut utiliser le mot-clé `global` suivi de la variable dans sa fonction.

```
In [29]: x = 2
         def incrémenter():
             global x
             x += 1

         incrémenter()
         print(x)
```
```
3
```

On voit qu'après avoir appliqué la fonction, la variable `x` ne vaut plus 2 mais 3 en dehors de la fonction.

Si une variable **locale** a le **même nom** qu'une variable **globale**, elle prend le dessus **dans la fonction**. En dehors de la fonction, la variable globale ne change **pas**.

```
In [30]: x = 2

         def puissance_3(n):
             x = 3
             return n**x

         puissance_3(5), x
```
```
Out[30]: (125, 2)
```