Python – Les fonctions

Sommaire

1. Fonctions

1.1 Principe et généralités

En programmation, les fonctions sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme. Elles permettent également de rendre le code plus lisible et plus clair en le fractionnant en blocs logiques.

1.2 Fonctions mathématiques (module math)

faire import math

1.2.1 Fonctions mathématiques :

- math.floor(-7.6) : partie entière, donne ici -8.0.
- int(math.floor(4.5)) : pour avoir l'entier 4.
- math.ceil(-7.6) : entier immédiatement supérieur, donne ici -7.
- math.exp(2) : exponentielle.
- math.log(2) : logarithme en base naturelle.
- math.log10(2) : logarithme en base 10.
- math.log(8, 2) : log de 8 en base 2.
- math.sqrt : racine carrée.
- math.pow(4, 5) : 4 puissance 5.
- math.fmod(4.7, 1.5) : modulo, ici 0.2. Préférer cette fonction à % pour les flottants.
- math.factorial(4) : factorielle 4, donc 24 (uniquement pour les entiers positifs).
- math.fsum(l) : fait la somme des éléments d'une liste, à préférer à sum car moins d'erreurs d'arrondis (comparer math.fsum([0.01 for i in range(100)]) et sum([0.01 for i in range(100)]))
- math.isinf(x) : teste si x est infini (inf) et renvoie True si c'est le cas.
- math.isnan(x) : teste si x est nan (Not a Number) et renvoie True si c'est le cas.
- math.gamma : la fonction gamma (pour n entier, gamma(n) = factorielle(n-1))
- math.erf : la fonction d'erreur (intégrale d'une gaussienne).

1.2.2 Fonctions trigonométriques :

- fonctions trigonométriques : math.sin, math.cos, math.tan, math.asin, math.acos, math.atan (l'argument est en radians).
- fonctions hyperboliques : math.sinh, math.cosh, math.tanh, math.asinh, math.acosh, math.tanh
- math.degrees(x) : convertit de radians en degrés (math.radians(x) pour l'inverse).
- math.pi, math.e : les constantes.

1.3 Définition d'une fonction

Pour définir une fonction, Python utilise le mot-clé def et si on veut que celle-ci renvoie une valeur, il faut utiliser le mot-clé return. Par exemple :

```
>>> def carre(x):
...     return x**2
...
>>> print(carre(2))
4
```

Notez que la syntaxe de def utilise les : comme les boucles for, while ainsi que les tests if, un bloc d'instrutions est donc attendu. De même que pour les boucles et les tests, l'indentation de ce bloc d'instructions (i.e. le corps de la fonction) est obligatoire.

Dans l'exemple précédent, nous avons passé un argument à la fonction carre() qui nous a retourné une valeur que nous avons affichée à l'écran. Que veut dire valeur retournée ? Et bien cela signifie que cette dernière est stockable dans une variable :

```
>>> res = carre(2)
>>> print(res)
4
```

Ici, le résultat renvoyé par la fonction est stockée dans la variable res.

1.4 Passage d'arguments

Une particularité des fonctions en Python est que vous n'êtes pas obligé de préciser le type des arguments que vous lui passez, dès lors que les opérations que vous effectuez avec ces arguments sont valides. Python est en effet connu comme étant un langage au typage dynamique, c'est-à-dire qu'il reconnaît pour vous le type des variables au moment de l'exécution, par exemple :

```
>>> def fois(x,y):
...     return x*y
...
>>> fois(2,3)
6
>>> fois(3.1415,5.23)
16.430045000000003
>>> fois('to',2)
'toto'
```

L'opérateur * reconnait plusieurs types (entiers, réels, chaînes de caractères), notre fonction est donc capable d'effectuer des tâches différentes ! Même si Python permet cela, méfiez-vous tout de même de cette grande flexibilité qui pourrait mener à des surprises dans vos futurs programmes. En général il est plus judicieux que chaque argument ait un type précis (int, str, float, etc), et pas l'un ou l'autre.
Un énorme avantage en Python est que les fonctions sont capables de renvoyer plusieurs valeurs à la fois, comme dans cette fraction de code :

```
>>> def carre_cube(x):
...     return x**2,x**3
...
>>> carre_cube(2)
(4, 8)
```

En réalité Python ne renvoie qu'un seul objet, mais celui-ci peut être séquentiel, c'est à dire contenir lui-même plusieurs objets. Dans notre exemple Python renvoie un objet de type tuple.
Une fonction pourrait tout autant renvoyer une liste :

```
>>> def carre_cube2(x):
...     return [x**2,x**3]
...
>>> carre_cube2(3)
[9, 27]
```

Renvoyer un tuple ou une liste de deux arguments (ou plus) est notamment très pratique en conjonction avec l'affectation multiple, par exemple :

```
>>> z1,z2 = carre_cube2(3)
>>> z1
9
>>> z2
27
```

1.5.Exercices

Exercice 1 :

On considère la suite définie par :

$$\begin{cases} u_0 = 2 \\ u_n = 3 * u_{n-1} - 1 \end{cases}$$

a. Ecrire une fonction itérative qui prend comme paramètre un entier naturel n et calcule le terme un de la suite en utilisant une boucle for.
b. Ecrire une fonction récursive qui prend comme paramètre un entier naturel n et calcule le terme un de la suite.
c. Ecrire une fonction qui renvoie une liste contenant les n premiers termes de la suite.
d. Ecrire une fonction qui, étant donnée un entier m, calcule l'indice du premier terme de la suite supérieur ou égal à m (exemple : si m = 30, la fonction retournera 3 car tous les termes d'indice inférieur à 3 sont plus petits que 30).

Exercice 2 :

Trois exemples de tableaux pour illustrer les définitions :

```
>>> a = [1,4,3]
>>> b = [1,0,0]
>>> c = [1,0,2]
```

Soit t un tableau d'entiers de taille n.

a. Ecrire une fonction sansDoublons(t) qui retourne True si le tableau d'entiers t est sans doublons (c'est à dire sans apparition multiple d'un élément), False sinon.
   Exemples :

```
>>> sansDoublons(a) : True
>>> sansDoublons(b) : False
>>> sansDoublons(c) : True
```

b. Ecrire une fonction interne(t) qui retourne True si ∀i ∈ [0; n[, t[i] ∈ [0; n[et retourne False sinon.
   Exemples :

```
>>> interne(a) : False
>>> interne(b) : True
>>> interne(c) : True
```