NSI - Première

Python - 4 - Fonctions

qkzk

Les fonctions

Une fonction est une portion de code qui peut être exécutée lorsqu'on en a besoin.

Utiliser une fonction se fait en deux temps :

1. définition de la fonction,
2. appel de la fonction.

Exemples

On a déjà rencontré plusieurs fonctions, en particulier `len`. Cette fonction renvoie (on dit parfois *retourne*) le nombre d'éléments d'une collection.

- la définition de la fonction `len` est faite par le programme Python lui même, nous n'avons pas à nous en charger.
- l'appel de la fonction se fait ainsi : `len(objet)`

Par exemple :

```
>>> nom = "Raymond"
>>> len(nom)
7
```

On a aussi rencontré la fonction `print`. Cette fonction **ne renvoie rien** mais affiche un objet dans la console.

```
>>> valeur = print("Bonjour !")
Bonjour
>>> valeur == None
True
```

La librairie `random` propose de nombreuses fonctions. Nous allons en étudier une :

```
>>> import random
>>> dir(ranom)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
'SystemRandom', 'TWOPI', '_Sequence', '_Set', '__all__', '__builtins__',
'__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', '_accumulate', '_acos', '_bisect', '_ceil', '_cos', '_e', '_exp',
'_floor', '_inst', '_log', '_os', '_pi', '_random', '_repeat', '_sha512',
'_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn',
'betavariate', 'choice', 'choices', 'expovariate', 'gammavariate', 'gauss',
'getrandbits', 'getstate', 'lognormvariate', 'normalvariate', 'paretovariate',
'randbytes', 'randint', 'random', 'randrange', 'sample', 'seed', 'setstate',
'shuffle', 'triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']
>>> help(random.randint)
randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

Que lit-on ?

- la notation `randint(a, b)` nous indique que cette fonction nécessite deux *paramètres*.
- La ligne suivante nous indique que la fonction renvoie un entier aléatoire entre `a` et `b` inclus.

Sans surprise :

```
>>> random.randint(3, 7)
5
>>> random.randint(3, 7)
3
>>> random.randint(3, 7)
4
>>> random.randint(3, 7)
5
```

On pourrait résumer cette fonction à son type :

```
random.randint(int, int) -> int
```

Elle prend deux entiers et renvoie un entier.

Nous allons apprendre à appeler, concevoir, documenter des fonctions.

Mais avant ça...

Que pourrait-on dire du type de `print` ?

```
print(any, any, any, ...) -> None
```

`print` accepte autant de paramètres qu'on le souhaite et ne renvoie *rien*.

Le fait qu'elle affiche quelque chose dans la console est *un effet de bord*.

Exercice 0

Relire les deux lignes précédentes jusqu'à ce que vous les connaissiez par coeur.

Appeler une fonction

Appeler une fonction consiste à l'introduire dans une instruction en respectant son *type*, c'est à dire ses paramètres d'entrée et de sortie.

Il convient donc de le connaître ! C'est à ça que sert l'aide de Python, à écrire convenablement les appels des fonctions et des objets afin d'éviter les erreurs.

Exercice 1

1. Documentez-vous sur la fonction `random.choice`. Que fait-elle ? Quel est son type ?
2. Les instructions suivantes sont-elles valables ?

```
>>> import random
>>> random.choice("salut")
>>> random.choice(salut)
>>> random.choice("s")
>>> random.choice(12345)
>>> random.choice("12345")
```

Définir une fonction

Avant de se lancer dans la syntaxe et ses subtilités : pourquoi aurait-on besoin de définir une fonction ?

Dès qu'un morceau de code réalise une tâche spécifique ou dès qu'il est répété plusieurs fois, on est amené à l'inclure dans une fonction.

Cela permet d'éviter de le recopier plusieurs fois (et donc d'avoir à le modifier autant de fois lorsqu'on voudra l'adapter).

Cela permet aussi de le tester indépendamment et donc de s'assurer qu'il fait ce qu'on attend de lui.

ce point peut sembler secondaire, pourquoi devrait-on "tester" le code. C'est en fait fondamental.

Imaginons devoir manipuler des températures. Il existe deux unités de mesure couramment employée : les degrés celcius et les Fahrenheit.

La conversion se fait à l'aide de la formule :

$$T_F = \frac{9}{5} \times T_C + 32$$

Bien sûr on pourrait faire :

```
>>> 100.0 * 9.0 / 5.0 + 32.0
212.0
>>> 37.0 * 9.0 / 5.0 + 32.0
98.6
>>> 233.0 * 9.0 / 5.0 - 32.0
387.4
```

et convertir la température à l'aide de la formule. Même avec un copier-coller cette approche est pénible... et dangereuse. Avez-vous repéré l'erreur dans les lignes précédentes ?

Définition d'une fonction : syntaxe

- On définit une fonction avec le mot clé `def`
- après `def` on trouve le nom de la fonction,
- ensuite viennent ses paramètres entre des parenthèses,
- ensuite le symbole `:`
- Après `def` on trouve un bloc indenté contenant le corps de la fonction.
- La fonction s'arrête après le mot clé `return` qui indique ce qu'elle renvoie ou lorsque l'indentation revient au niveau précédent.

```
def fahrenheit(degre_celcius):
  """Converti une temperature des degrés celcius vers les Fahrenheit"""
  return degre_celcius * 9.0 / 5.0 + 32.0
```

On peut maintenant l'utiliser comme ceci :

```
>>> fahrenheit(100)
212.0
>>> fahrenheit(37)
98.6
```

La ligne `"""converti une temperature des degrés celcius vers les Fahrenheit"""` appelée *docstring* ou documentation est une chaîne de caractères pouvant occuper plusieurs lignes.

On peut même consulter sa documentation :

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

fahrenheit(degre_celsius)
    Converti une temperature des degrés celcius vers les Fahrenheit
```

Quelques recommandations importantes :

- chaque fois qu'on est amené à effectuer plusieurs manipulations similaires on crée une fonction,
- **une bonne fonction ne fait qu'une chose**,
- toutes vos fonctions doivent être documentées,
- on évite, autant que possible, les effets de bord.

Résumé : le mot clé `def`

```
def nom_de_la_fonction(parametre1, parametre2, parametre3, ...):
    """ Documentation qu'on peut écrire
    sur plusieurs lignes
    """
    bloc d instructions     # attention à l'indentation

    return resultat         # valeur de sortie

arrive ici la fonction est finie
```

Exercice 2 : documentation

Pour chaque fonction définie ci-dessous :

1. Indiquer son type d'entrée et de sortie
2. A-t-elle un effet de bord ?
3. Compléter sa documentation et proposer un nom plus adapté

```
def ma_fonction():
  ''' '''
  return 3

def seconde_fonction(nombre):
  ''' '''
  return 5 * nombre

def troisieme_fonction(nom):
  ''' '''
  print("Bonjour", nom)
```

Exercice 3 : dé à n faces

La fonction suivante simule le lancer d'un dé à six faces :

```
import random


def lancer_un_de():
    '''Renvoie un entier aléatoire entre 1 et 6'''
    valeur = random.randint(1, 6)
    return valeur
```

On l'exécute ainsi :

```
>>> lancer_un_de()
3
>>> lancer_un_de()
6
```

Adapter la fonction pour qu'elle accepte un paramètre, le nombre de faces du dé.

On souhaite pouvoir l'utiliser ainsi :

```
>>> lancer_un_de(20)
14
>>> lancer_un_de(10)
7
```

Syntaxe complète et documentation complète

Voici une expression mathématiques : $f(x) = 2x + 1$

On peut la traduire simplement en :

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

Ce n'est pas très lisible et l'aide ne va pas nous apporter grand chose.

Donnons lui un nom explicite

```
def image_de_f(x):
    return 2 * x + 1
```

Ajoutons sa documentation

```
def image_de_f(x):
    '''Renvoie l'image de x par f(x) = 2x + 1'''
    return 2 * x + 1
```

On peut aussi ajouter des *indications de type*, ce n'est pas obligatoire mais c'est pratique !

```
def image_de_f(x: float) -> float:
    '''Renvoie l'image de x par f(x) = 2x + 1'''
    return 2 * x + 1
```

Remarquez la syntaxe :

- dans les parenthèses on met `x: float` c'est le type du paramètre `x`,
- derrnière les parenthèses on met : `-> float` c'est le type de sortie.

On peut indiquer aussi les types dans la documentation :

```
def image_de_f(x: float) -> float:
    '''
    Renvoie l'image de x par f(x) = 2x + 1

    @param x: (float) l'antécédent
    @return:  (float) l'image
    '''
    return 2 * x + 1
```

et maintenant :

```
>>> image_de_f(31)
63
>>> help(image_de_f)
Help on function image_fonction in module __main__:

image_de_f(x: float) -> float
    Renvoie l'image de x par f(x) = 2x + 1
    @param x: (float) l'antécédent
    @return:  (float) l'image
```

Exercice 4 - code d'une fonction

Pour chaque question on donnera le code complet avec documentation et indication de type.

Ne faites pas tout d'un coup, suivez la démarche proposée dans le paragraphe précédent.

1. Écrire une fonction qui calcule l'image par $f(x) = x^2 - 2x + 3$
2. Les fonctions peuvent accepter plusieurs paramètres d'entrée. Écrire une fonction qui prend trois paramètres entiers et renvoie leur somme.

   ```
   >>> somme_de_trois_nombres(1, 2, 3)
   6
   ```

3. `contient_un_z` accepte une chaîne de caractère en paramètre d'entrée et renvoie un booléen. Il est vrai si la chaîne reçue contient la lettre `z`
4. `contient_la_lettre` accepte *deux* paramètres d'entrée, des chaînes, elle renvoie vraie si le second paramètre (une lettre) est présent dans le premier.
5. `longueur` simule le comportement de `len` pour les chaînes de caractères.

   C'est évident mais je le précise, vous n'avez pas le droit d'utiliser `len` !
6. `factorielle` renvoie le produit des entiers jusqu'au paramètre `n`, un entier plus grand que 0.

   ```
   >>> factorielle(5)
   120
   ```

   Complément: Modifier la fonction pour respecter la définition mathématique: `factorielle(0) == 1`
7. `plus_grand_des_deux` prend deux paramètres entiers et renvoie le plus grand des deux.

Exercice 5 : simplifier le code d'une fonction

C'est une démarche qu'il faut envisager systématiquement : nos fonctions doivent être les plus simples possibles.

Commençons par un exemple simple : la fonction `est_pair` accepte un entier en parametre et renvoie un booléen vrai si l'entier est pair, faux sinon.

```
def est_pair(entier):
    if entier % 2 == 0:
        return True
    else:
        return False
```

1. Lire et tester la fonction. Remarquez que le mot clé `return` est présent deux fois.
2. Ajouter la documentation et les indications de type.
3. Quelles sont les valeurs possibles de `entier % 2` ?
4. Quelles sont les valeurs possibles de `entier % 2 == 0` ? Dans quel cas est-il égal à `True` ? à `False` ?

On en déduit qu'il est possible d'écrire la fonction ainsi :

```
def est_pair_plus_court(entier):
    return entier % 2 == 0
```

5. Vérifiez sur des exemples que les deux versions font la même chose.

Portée des variables : variables globales et locales

La *portée* d'une variable et la partie du programme dans laquelle cette variable est définie.

On distingue essentiellement deux types de portées : les variables *globales* qui sont accessibles partout et les variables *locales* qui n'existent que dans une fonction.

Considérons l'exemple suivant :

```
a = 10          # a est une variable GLOBALE

def ma_fonction():

    return a
```

Exécutons le script

Lorsqu'on exécute `ma_fonction()` elle renvoie 10.

```
>>> a               # nous sommes dans l'espace GLOBAL
10
>>> ma_fonction()   # ma_fonction a accès a cet espace
10
```

Et maintenant :

```
a = 10          # a est une variable GLOBALE

def ma_fonction():

    a = 20      # on défini une variable LOCALE
    return a
```

```
>>> a               # a vaut toujours 10 :)
10
>>> ma_fonction()   # ma_fonction renvoie SA VALEUR de a
20
>>> a               # la valeur GLOBALE de a n'a pas changé !
10
```

Le mot clé `global`

Le mot clé `global`, indiqué au début du bloc de code d'une fonction, précise qu'une variable est maintenant globale et qu'on peut la modifier.

```python
a = 10          # a est une variable GLOBALE

def ma_fonction():

    global a    # a est maintenant globale !

    a = 20      # on défini une variable LOCALE
    return a
```

```
>>> a               # a vaut toujours 10 :)
10
>>> ma_fonction()   # ma_fonction renvoie SA VALEUR de a, MAIS !!!
20
>>> a               # la valeur GLOBALE A ETE CHANGEE PAR ma_fonction
20
```

L'utilisation de variables globales est une mauvaise pratique.

Il est difficile de suivre l'évolution de variables globales lorsqu'on lit un script.

Exercices

Pour chaque fonction produite on attend :

- des noms explicites,
- la documentation complète,
- des exemples qui attestent que la fonction fait ce qui est attendu,

Exercice 6

Écrire une fonction `carre` qui calcule le carré d'un nombre.

Écrire une fonction qui calcule l'aire d'un disque.

```
>>> from math import pi
>>> pi
3.141592653589793
```

Rappels aux grands géomètres que vous êtes : l'aire d'un disque de rayon R est donnée par :

$$\mathcal{A} = \pi R^2$$

Exercice 7

Écrire une fonction qui prend deux chaînes de caractères et renvoie la plus courte des deux.

```
>>> plus_court("abc", "abcd")
'abc'
>>> plus_court("abcd", "ab")
'ab'
```

Exercice 8 - la fonction `sum`

Python propose de nombreuses fonctions très pratiques... pourquoi les réécrire ?

Et bien pour apprendre c'est un excellent exercice !

1. Que fait la fonction `sum` ?

2. Comment calculer la somme des entiers de 1 à 100 avec la fonction `sum` ?
3. Voici un algorithme pour calculer la somme des entiers entre a et b inclus :

```
somme = 0
Pour k allant de a à b inclus,
    somme = somme + k
fin du pour

# à cette étape, somme contient la valeur !
```

Programmer cet algorithme dans une fonction `somme` qui accepte deux paramètres `a` et `b` et renvoie la somme des entiers entre `a` et `b`.

On prendra garde aux noms, dans l'algorithme `somme` désigne un *nombre* mais pour notre fonction, `somme` est *la fonction elle-même !*, il faudra donc employer un *autre nom* pour désigner la valeur de la somme.

Exercice 9 - générateur de mot de passe

1. Relisez l'aide de la fonction `random.choice`.
2. À l'aide de cette fonction, créer une fonction `password` qui prend un paramètre entier (la taille du mot de passe) et renvoie un mot de passe constitué de lettres ou de chiffres respectant la taille donnée.

```
symboles = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'`
```

```
>>> password(10)
'mHVeC5rs8P'
>>> password(6)
'PYthoN'
```

Exercice 10 - déterminer la sortie

Pour chaque question suivante, vous analysez le code écrit, prédisez la sortie et vérifiez. Faites le sérieusement sinon vous aurez de grandes surprises en devoir...

1. Qu'affiche le script suivant ?

```
def func(a):
    a += 2.0
    return a

a = func(8.0)
print(a)
```

2. Qu'affiche le script suivant ?

```
def diff(val1,val2):
    return val2 - val1

a = diff(3.0,-2.0)
print(a)
```

3. Qu'affiche le script suivant ?

```
def func(val):
  if val < 0.0:
      return 0
  return val

a = func(-1.5)
print(a)
```

4. Qu'affiche le script suivant ?

```
def carre(val):
    return val*val

def inc(val):
    return val + 1

a = carre(inc(3.0))
print(a)
```

5. Qu'affiche le script suivant ?

```
def func(a):
  a += 2.0
  return a

a = 5.0
b = func(a)
print(a, b)
```

6. Qu'affiche le script suivant ?

```
def func(a):
    a += 7

a = 9
b = func(a)
print(a, b)
```

7. *Plus difficile.* Qu'affiche le script suivant ?

```
def f(x):
    return x

y = f(4)
print(y)
g = f(f)
print(g(2))
f = f(3)
print(f)
print(g(5))
print(f(6))
```