Introduction à la Programmation des Algorithmes

5.3. Python – Fonctions et portées

François Fleuret

https://fleuret.org/11x001/

UNIVERSITÉ
DE GENÈVE

Comme en C, Python permet de définir des fonctions qui peuvent recevoir des arguments et retourner des valeurs.

Une définition de fonction se fait avec `def`.

Comme toujours en Python, les types sont laissés libres, aussi bien pour les arguments que pour les valeurs retournées.

```python
def le_plus_grand(a, b):
    if a >= b:
        return a
    else:
        return b

print(le_plus_grand(11, 15))
print(le_plus_grand(11, 15.3))
print(le_plus_grand("avion", "bateau"))
```

affiche

```
15
15.3
bateau
```

Les fonctions en Python sont donc capables de traiter des types différents quand les opérations qui les composent le sont.

Si les types des arguments passés au moment de l'appel de la fonction ne sont pas adéquats, l'erreur se produira dynamiquement au moment où une opération impossible sera tentée.

```python
def le_plus_grand(a, b):
    if a >= b:
        return a
    else:
        return b

print(le_plus_grand([ 11 ], 15.3))
```

affiche

```
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print(le_plus_grand([ 11 ], 15))
  File "test.py", line 2, in le_plus_grand
    if a >= b:
TypeError: '>=' not supported between instances of 'list' and 'float'
```

La première instruction d'une fonction peut, de façon facultative, être une chaîne de caractères littérale. Ce sera alors la chaîne de documentation de la fonction, appelée docstring.

```
1  def bonjour(nom):
2      """Dir bonjour à quelq'un.
3
4      Affiche un "bonjour" suivi du nom fourni en argument.
5      """
6
7      print("Bonjour", nom)
```

Portées

Comme en C et dans tous les langages de programmation, un identifiant en Python existe et fait référence à une quantité donnée dans une sous-partie du programme que l'on appelle sa **portée**.

La différence majeure entre le C et le Python est que pour le premier on parle de **portée lexicale**, et pour le second de **portée dynamique**.

Comme en C et dans tous les langages de programmation, un identifiant en Python existe et fait référence à une quantité donnée dans une sous-partie du programme que l'on appelle sa **portée**.

La différence majeure entre le C et le Python est que pour le premier on parle de **portée lexicale**, et pour le second de **portée dynamique**.

Dans le cas de la portée lexicale, un identifiant existe dans une portion du code source qui ne dépend pas de l'exécution du programme, simplement de la forme du code source (cf. 2.3. "Langage C – Portée et variables locales").

Comme en C et dans tous les langages de programmation, un identifiant en Python existe et fait référence à une quantité donnée dans une sous-partie du programme que l'on appelle sa **portée**.

La différence majeure entre le C et le Python est que pour le premier on parle de **portée lexicale**, et pour le second de **portée dynamique**.

Dans le cas de la portée lexicale, un identifiant existe dans une portion du code source qui ne dépend pas de l'exécution du programme, simplement de la forme du code source (cf. 2.3. "Langage C – Portée et variables locales").

Avec la portée dynamique, un identifiant existe selon l'exécution du programme et n'est pas limité à une partie du source clairement identifiée.

La portée dynamique offre plus de souplesse, mais amène à plus d'erreurs de programmation difficiles à trouver.

Il existe en Python des **variables globales**, créées en dehors des fonctions, et des **variables locales** qui sont propres à une fonction.

Python crée une variable locale à une fonction dès que l'on affecte une valeur à un identifiant dans le corps de la fonction.

Dans le corps d'une fonction, un identifiant fait référence en priorité à la variable locale si elle existe, sinon à la variable globale.

Il n'y a ici pas d'affectation dans la fonction `truc`, donc l'identifiant `s` dans le corps de cette fonction fait référence à la variable globale.

```
1  def truc():
2      print('dans truc:', s)
3
4  s = 'Valeur globale'
5
6  print('dehors avant:', s)
7  truc()
8  print('dehors apres:', s)
```

affiche

```
dehors avant: Valeur globale
dans truc: Valeur globale
dehors apres: Valeur globale
```

Il y a ici une affectation dans la fonction `truc`, donc l'identifiant `s` dans le corps de cette fonction fait référence à la variable locale ainsi créée.

```
1  def truc():
2      s = 'Valeur locale'
3      print('dans truc:', s)
4
5  s = 'Valeur globale'
6
7  print('dehors avant:', s)
8  truc()
9  print('dehors apres:', s)
```

affiche

```
dehors avant: Valeur globale
dans truc: Valeur locale
dehors apres: Valeur globale
```

Le mot clé `global` dans le corps de la fonction `truc` indique que l'identifiant `s` fait référence à la variable globale et que l'affectation ne doit pas créer de variable locale.

```
1   def truc():
2       global s
3       s = 'Valeur locale'
4       print('dans truc:', s)
5
6   s = 'Valeur globale'
7
8   print('dehors avant:', s)
9   truc()
10  print('dehors apres:', s)
```

affiche

```
dehors avant: Valeur globale
dans truc: Valeur locale
dehors apres: Valeur locale
```

Ici l'identifiant s est utilisé avant qu'il y ait une affectation, ce qui crée une possible ambiguïté et provoque une erreur.

```
1  def truc():
2      print('dans truc:', s)
3      s = 'Valeur locale'
4
5  s = 'Valeur globale'
6
7  print('dehors avant:', s)
8  truc()
9  print('dehors apres:', s)
```

affiche

```
dehors avant: Valeur globale
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    truc()
  File "test.py", line 2, in truc
    print('dans truc:', s)
UnboundLocalError: local variable 's' referenced before assignment
```

Cette ambiguïté est levée par le mot clé `global`.

```
1   def truc():
2       global s
3       print('dans truc:', s)
4       s = 'Valeur locale'
5
6   s = 'Valeur globale'
7
8   print('dehors avant:', s)
9   truc()
10  print('dehors apres:', s)
```

affiche

```
dehors avant: Valeur globale
dans truc: Valeur globale
dehors apres: Valeur locale
```

Une affectation dans la clause d'une condition ou d'une boucle ne crée pas de variable locale

```
1  s = 1
2  print('debut', s)
3
4  for t in range(30):
5      if t%5 == 0:
6          s = t
7
8  print('fin', s)
```

affiche

```
debut 1
fin 25
```

Contrairement au C, Python permet de définir des fonctions locales à une fonction.

```
1   def truc():
2       def chose():
3           print('chose', s)
4
5       s = 'Second'
6       print('truc', s)
7       chose()
8
9   s = 'Premier'
10
11  truc()
12  print('global', s)
```

affiche

```
truc Second
chose Second
global Premier
```

Arguments d'une fonction

Il est possible en python de donner des valeurs par défaut à des arguments d'une fonction, auquel cas il n'est pas obligatoire de spécifier les arguments correspondant au moment de l'appel.

```
1  def tete(l, t = 1):
2      return l[0:t]
3
4  print(tete([ 1, 2, 3, 4, 5, 6, 7, 8 ]))
5
6  print(tete([ 1, 2, 3, 4, 5, 6, 7, 8 ], 3))
```

affiche

```
[1]
[1, 2, 3]
```

Un argument sans valeur par défaut ne peut pas suivre un argument avec.

```
1   def tete(l, t = 1, u):
2       return l[0:t] + u
```

affiche

```
  File "test.py", line 1
    def tete(l, t = 1, u):
             ^
SyntaxError: non-default argument follows default argument
```

Python permet de spécifier les argument en utilisant leur identifiant plutôt que leur ordre.

```
1   def boite(hauteur, largeur):
2       s = ''
3       for i in range(hauteur):
4           for j in range(largeur):
5               s += '*'
6           s += '\n'
7       return s
8
9   print(boite(3, 5))
10  print(boite(largeur = 3, hauteur = 5))
```

affiche

```
*****
*****
*****

***
***
***
***
***
```

C'est particulièrement utile s'il y a un grand nombre d'arguments avec des valeurs par défaut.

```
1  def boite(hauteur = 2, largeur = 6, symbole = '+'):
2      s = ''
3      for i in range(hauteur):
4          for j in range(largeur):
5              s += symbole
6          s += '\n'
7      return s
8
9  print(boite(largeur = 3))
10 print(boite(3, symbole = 'x'))
```

affiche

```
+++
+++

xxxxxx
xxxxxx
xxxxxx
```

Il peut arriver que l'on veuille écrire une fonction qui reçoit un nombre indéfini d'arguments en plus de ceux spécifiés de façon classique.

Il faut rajouter un argument en faisant précéder son identifiant de * pour obtenir un tuple avec les arguments supplémentaires non nommés, et un autre précédés de ** pour obtenir un dictionnaire des arguments supplémentaires nommés.

```
1  def truc(a, b, *args, **kargs):
2      print('a', a)
3      print('b', b)
4      print('args', args)
5      print('kargs', kargs)
6
7  truc(1, 2, 3, 4, 5, chose = 4, coucou = 'haha')
```

affiche

```
a 1
b 2
args (3, 4, 5)
kargs {'chose': 4, 'coucou': 'haha'}
```

Finalement, il arrive que l'on veuille passer en arguments à une fonction des valeurs contenue dans un tuple ou une liste. L'opérateur * permet de séparer les éléments comme autant d'argument.

```
1  def machin(a, b, c):
2      print('a', a)
3      print('b', b)
4      print('c', c)
5
6  v = [ 11, 22, '33' ]
7  machin(*v)
```

affiche

```
a 11
b 22
c 33
```

L'opérateur ** fait de même pour séparer un dictionnaire en arguments nommés.

```
1  def machin(a = 1, b = 2, c = 3):
2      print('a', a)
3      print('b', b)
4      print('c', c)
5
6  v = { 'b': 11 }
7  machin(**v)
```

affiche

```
a 1
b 11
c 3
```

Fin