Introduction à la Programmation des Algorithmes

5.4. Python – Programmation fonctionnelle

François Fleuret

https://fleuret.org/11x001/

UNIVERSITÉ
DE GENÈVE

Python est un langage **fonctionnel** dans lequel une fonction est une valeur qui peut être copiée dans une variable, et plus généralement manipulée, au même titre que des grandeurs numériques.

```
1   def fois_deux(n):
2       return 2 * n
3
4   def plus_un(n):
5       return n + 1
6
7   print(fois_deux(10), plus_un(5))
8
9   f = plus_un
10  print(f(3))
11
12  f = fois_deux
13  print(f(3))
```

affiche

```
20 6
4
6
```

Les fonctions étant des valeurs à part entière, on peut en particulier les utiliser dans des types composés comme des listes.

```
1   def fois_deux(n):
2       return 2 * n
3
4   def plus_un(n):
5       return n + 1
6
7   n = 10
8   for f in [ fois_deux, plus_un, plus_un ]:
9       n = f(n)
10  print(n)
```

affiche

22

Fonctions d'ordres supérieurs

Les fonctions peuvent également être passées en arguments, ou renvoyées comme résultat d'autres fonctions. On parle alors pour ces dernières de **fonctions d'ordres supérieurs.**

```
1  def fois_deux(n):
2      return 2 * n
3
4  def plus_un(n):
5      return n + 1
6
7  def applique_deux(n, f, g):
8      return g(f(n))
9
10 print(applique_deux(5, plus_un, plus_un))
11
12 print(applique_deux(5, plus_un, fois_deux))
```

affiche

```
7
12
```

```
1   def fois_deux(n):
2       return 2 * n
3
4   def plus_un(n):
5       return n + 1
6
7   def applique_plusieurs(n, fs):
8       for f in fs:
9           n = f(n)
10      return n
11
12  print(applique_plusieurs(10, [ fois_deux, plus_un, plus_un ]))
```

affiche

22

```python
def resoud(a, b, f, epsilon = 1e-6):
    """Retourne la racine de f entre a et b.

La fonction f doit être croissante entre ces deux valeurs.
"""

    while b - a >= epsilon:
        c = (a + b) / 2
        if (f(c) >= 0):
            b = c
        else:
            a = c

    return c
```

```python
def carre_moins_deux(x):
    return x * x - 2

print(resoud(0, 2, carre_moins_deux))
```

affiche

```
1.4142141342163086
```

Python permet en particulier de retourner comme résultat une fonction définie localement.

```
1  def ajouteur(k):
2      def f(n):
3          return n + k
4      return f
5
6  ajoute_cinq = ajouteur(5)
7
8  print(ajoute_cinq(98))
```

affiche

103

```
1  def ajouteur(k):
2      def f(n):
3          return n + k
4      return f
5
6  ajoute_cinq = ajouteur(5)
7
8  print(ajoute_cinq(98))
```

Un point technique remarquable est que le `k` passé en argument ligne 6 est toujours accessible ligne 8.

Cela est possible car Python garde automatiquement dans l'objet retourné par `ajouteur` une copie de toutes les variables locales qui existaient quand `f` a été définie. La combinaison de la fonction et de ce contexte est appelée une *closure*.

En pratique cette machinerie complexe est assez bien faite pour que son fonctionnement soit intuitif.

```
1  def compose(f, g):
2      def h(n):
3          return g(f(n))
4      return h
5
6  psi = compose(fois_deux, plus_un)
7  print(psi(3))
```

affiche

7

```python
def pol_from_coeffs(alphas):

    def pol(x):
        s, y = 0, 1
        for a in alphas:
            s += a * y
            y *= x
        return s

    return pol

f = pol_from_coeffs([ -1, 0, 1 ])

print(f(2), f(10.0))
```

affiche

```
3 99.0
```

Les variables globales sont accessibles depuis la fonction d'une closure.

```
1   def truc():
2       def f(n):
3           return n + b
4       return f
5
6   h = truc()
7
8   b = 1
9   print(h(2))
10
11  b = 10
12  print(h(2))
```

affiche

```
3
12
```

Fonctions anonymes

Il arrive souvent que l'on ait besoin de définir des fonctions pour ne les utiliser qu'une fois.

```
1   def fois_deux(n):
2       return 2 * n
3
4   def plus_un(n):
5       return n + 1
6
7   def applique_deux(n, f, g):
8       return g(f(n))
9
10  print(applique_deux(5, plus_un, fois_deux))
```

Python permet de noter des **fonctions anonymes** (*lambda functions*) avec une syntaxe concise, et auxquelles n'est pas associé d'identifiant. Ces fonctions sont limitées au calcul d'une expression.

```
lambda var_1, var_2, ..., var_n: expression
```

Par exemple

```
1  def truc(x):
2      return x + 1
3
4  f = truc
```

et

```
1  f = lambda x: x + 1
```

sont équivalents.

```python
def machin(x):
    return x * 100

truc = lambda x: x * 10 + x

print(machin(5), truc(5))
```

affiche

```
500 55
```

```
1  def applique_deux(n, f, g):
2      return g(f(n))
3
4  print(applique_deux(5, lambda n: n + 1, lambda n: 2 * n))
```

affiche

12

La syntaxe de Python permet d'appliquer une fonction anonyme directement, sans qu'elle soit dans une variable.

```
1  print((lambda n: n + 3)(120))
2  print((lambda x: (x-1, x, x+1))(120))
3  print((lambda a, b, c: a * b + c)(10, 5, 4))
```

affiche

```
123
(119, 120, 121)
54
```

La syntaxe des fonctions anonymes permet de bien comprendre un appel de fonction dans une évaluation par substitution.

Avec `a = 2`, `b = 3`, et `f = lambda x, y: x * y`:

```
f(a + b, a - b)
f(2 + b, a - b)
f(2 + 3, a - b)
f(5, a - b)
f(5, 2 - b)
f(5, 2 - 3)
f(5, -1)
(lambda x, y: x * y)(5, -1)
5 * (-1)
-5
```

Une fonction anonyme peut aussi être dans une closure.

```
1  def quadratique(a, b, c):
2      return lambda x: a * x * x + b * x + c
3
4  f = quadratique(1, 2, 1)
5
6  print(f(5))
```

affiche

36

On peut toujours exprimer une fonction à plusieurs arguments avec des fonctions d'ordres supérieurs à un seul argument (on appelle cette réduction le *"currying"* en hommage à Haskell Curry).

Par exemple:

```
>>> (lambda n, m: m//n)(10, 50)
5
>>> (lambda n: lambda m: m//n)(10)(50)
5
>>> (lambda a, b, c: a + b * c)(1, 2, 3)
7
>>> (lambda a: lambda b: lambda c: a + b * c)(1)(2)(3)
7
```

Opérateur conditionnel ternaire

Python offre une construction pour des expressions conditionnelles qui repose sur un opérateur conditionnel ternaire

```
expression1 if condition else expression2
```

Python offre une construction pour des expressions conditionnelles qui repose sur un opérateur conditionnel ternaire

```
expression1 if condition else expression2
```

Par exemple

```
1  >>> 4 if 2 < 3 else 5
2  4
3  >>> 14 + 4 if 3 < 2 else 11
4  11
```

Python offre une construction pour des expressions conditionnelles qui repose sur un opérateur conditionnel ternaire

```
expression1 if condition else expression2
```

Par exemple

```
1  >>> 4 if 2 < 3 else 5
2  4
3  >>> 14 + 4 if 3 < 2 else 11
4  11
```

Bien que cet opérateur utilise les mots clés `if` et `else`, il n'a rien à voir avec l'instruction `if` de contrôle de flux.

Cet opérateur est particulièrement utile pour définir des conditions dans des fonctions anonymes, puisque ces dernières doivent se limiter à des expressions.

```
1  le_plus_gros = lambda a, b: a if a >= b else b
2
3  print(le_plus_gros(3, 4.5))
```

affiche

Cet opérateur est particulièrement utile pour définir des conditions dans des fonctions anonymes, puisque ces dernières doivent se limiter à des expressions.

```
1  le_plus_gros = lambda a, b: a if a >= b else b
2
3  print(le_plus_gros(3, 4.5))
```

affiche

```
4.5
```

```
1  le_plus_gros = lambda a, b: a if a >= b else b
```

Par substitution, nous avons

```
le_plus_gros(3, 4.5)
(lambda a, b: a if a >= b else b)(3, 4.5)
3 if 3 >= 4.5 else 4.5
3 if False else 4.5
4.5
```

Fin