

Séance 6

Programmation fonctionnelle, décorateur et générateurs



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

Objectifs

- Bases de la **programmation fonctionnelle**
 - Définition et utilisation de fonctions
 - Fonction lambda
- Création et utilisation de **décorateurs**
 - Définition d'un décorateur
 - Closure
- Création et utilisation de **générateurs**
 - Instruction yield



f(x) =

Programmation fonctionnelle

Fonction

- Une **fonction** renvoie une valeur sur base de paramètres

Comme on les définit en mathématiques

- **Caractérisation** d'une fonction

- Nom
- Liste des paramètres avec leurs types
- Valeur de retour et son type

```
1 def add1(n):  
2     return n + 1  
3  
4 print(add1(7))
```

Objet fonction (1)

- Une fonction est un **objet de première classe**

Peut se passer en paramètre comme n'importe quelle autre valeur

- Une fonction est un **objet** de type function

```
1 def add1(n):
2     return n + 1
3
4 print(add1)
5 print(type(add1))
```

```
<function add1 at 0x10070eea0>
<class 'function'>
```

Objet fonction (2)

- On peut **renvoyer une fonction** comme valeur de retour

On fait référence à une fonction à partir de son nom

```
1 def add1(n):
2     return n + 1
3
4 def givemefun():
5     return add1
6
7 f = givemefun()
8 print(f(41))
```

Fonction imbriquée (1)

- On peut **imbriquer une fonction** dans un autre fonction

La fonction imbriquée est définie dans le corps d'une autre

- On peut **envoyer une fonction imbriquée**

```
1 def buildadder():
2     def add1(n):
3         return n + 1
4     return add1
5
6 f = buildadder()
7 print(f(41))
```

Fonction imbriquée (2)

- La fonction imbriquée est **invisible** en dehors du parent

Ne peut être appelée que depuis le parent

```
1 def buildadder():
2     def add1(n):
3         return n + 1
4     return add1
5
6 print(add1(41))
```

```
Traceback (most recent call last):
  File "test.py", line 6, in <module>
    print(add1(41))
NameError: name 'add1' is not defined
```

Paramètres positionnel

- Récupération des **paramètres positionnels** avec *args

La valeur est un tuple reprenant les paramètres dans l'ordre

```
1 def f(*args):  
2     print(args)  
3  
4 f(32)  
5 f('12', 98)
```

```
(32,)  
[]  
('12', 98)  
[]
```

Paramètre optionnel

- Récupération des **paramètres optionnels** avec **kwargs

La valeur est un dictionnaire reprenant les paramètres

```
1 def f(*args, **kwargs):  
2     print(args)  
3     print(kwargs)  
4  
5 f(name=99)  
6 f(0, ready=True)
```

```
()  
{'name': 99}  
(0,)  
{'ready': True}
```

Fonction d'ordre supérieur

- Fonction **d'ordre supérieur** ont les propriétés suivantes
 - prendre une ou plusieurs fonctions en paramètres
 - et/ou renvoyer une fonction
- Correspond à des **opérateurs** en mathématiques

Comme la dérivée par exemple

Fonction lambda (1)

- Une fonction lambda est une **fonction anonyme**

Elle ne possède pas de nom

- Déclaration à l'aide du **mot réservé lambda**

lambda paramètres : expression

```
1 compute = lambda n : n ** 2
2
3 print(compute(9))
```

Fonction lambda (2)

- Le corps d'une fonction lambda doit être **une expression**

Pas d'affectation ou instructions comme while, try...



- Utilisées lorsqu'on doit déclarer une fonction en **one-shot**

La fonction a une durée de vie limitée



```
1 def apply(f, n):
2     return f(n)
3
4 print(apply(lambda x : x + 1, 41))
5 print(apply(lambda x : x ** 2, 9))
```

Refactoring d'une fonction lambda

- Réfactorer une fonction lambda est complexe à comprendre

Recette de refactoring proposée par Fredrik Lundh

- Procédure de refactoring

- 1 Écrivez un commentaire qui décrit ce que fait la lamnda
- 2 Trouvez un nom qui capture l'essence du commentaire
- 3 Convertissez la lambda en une définition classique def
- 4 Retirez le commentaire

- Une fonction lambda n'est qu'un **sucré syntaxique**

Fonction map

- Application d'une fonction à tous les éléments d'une séquence
map calcule une nouvelle séquence et la renvoie
- Résultat renvoyé sous la forme d'un générateur
Utilisation de la fonction list pour le convertir

```
1 data = [1, 2, 3]
2 result = map(lambda x : x ** 2, data)
3 print(list(result))
```



```
[1, 4, 9]
```

Fonction filter

- Garde les éléments d'une séquence satisfaisant une condition
filter calcule une nouvelle séquence et la renvoie
- Résultat renvoyé sous la forme d'un générateur
Utilisation de la fonction list pour le convertir

```
1 data = [1, -2, 0, 7, -3]
2 result = filter(lambda x : x >= 0, data)
3 print(list(result))
```



```
[1, 0, 7]
```

Fonction sum

- Somme les éléments d'une séquence

L'opérateur + doit être défini sur les éléments de la séquence

- Opération de type réduction d'une séquence vers une valeur

Famille des opérations de type reduce

```
1 data = [2+1j, -1j, 4]
2 print(sum(data))
```



(6+0j)

Fonctions all et any

- Tous les/au moins un élément d'une séquence **vaut True**

Les fonctions all et any renvoient un booléen

```
1 data = [2, -6, 0, 8]
2
3 test = lambda e : e % 2 == 0 and e >= 0
4 filtered = [test(e) for e in data]
5 print(all(filtered))
6 print(any(filtered))
```



False
True

Type callable

- Un objet **callable** peut être appelé avec l'opérateur ()

Par exemple une fonction, méthode, classe...

- Test de “callabilité” avec la **fonction callable**

Fonction prédéfinie qui renvoie un booléen

```
1 class Empty:  
2     pass  
3  
4 def create():  
5     return Empty()  
6  
7 print(callable(Empty))      # True  
8 print(callable(create))    # True  
9 print(callable(callable))  # True
```

Les sept types callables

- Fonctions définies par l'utilisateur
- Fonctions prédéfinies
- Méthodes prédéfinies
- Méthodes définies dans une classe
- Classes
- Instance d'une classe
- Fonctions génératrices

Instances callables

- Une **instance d'une classe** peut être callable

Il suffit de redéfinir la méthode `__call__`

- Utile pour stocker de l'information entre deux appels

```
1 class Die:
2     def __init__(self):
3         self.roll()
4
5     @property
6     def value(self):
7         return self.__value
8
9     def roll(self):
10        self.__value = random.randint(1, 6)
11
12    def __call__(self):
13        self.roll()
14        return self.value
15
16 die = Die()
17 print(die.value)
18 print(die())
```

Closure

- Fonction avec une portée étendue à des variables non globales

Ces variables sont appelées variables libres

```
1 def make_adder(n):
2     def adder(value):
3         return value + n
4     return adder
5
6 adder2 = make_adder(2)
7 adder7 = make_adder(7)
8
9 print(adder2(12))
10 print(adder2(9))
```



14
11

Calcul incrémental d'une moyenne (1)

■ Stockage d'une séquence de valeurs

Recalcule de la moyenne à chaque appel à la fonction

```
1 def make_averager():
2     data = []
3     def averager(value):
4         data.append(value)
5         return sum(data) / len(data)
6     return averager
7
8 avg = make_averager()
9 print(avg(5))
10 print(avg(7))
11 print(avg(6))
```



```
5.0
6.0
6.0
```

Calcul incrémental d'une moyenne (2)

- Mot réservé `nonlocal` pour accéder variable libre

Seulement nécessaire en cas d'accès en modification à la variable

```
1 def make_averager():
2     total = 0
3     count = 0
4     def averager(value):
5         nonlocal total, count
6         total += value
7         count += 1
8         return total / count
9     return averager
10
11 avg = make_averager()
12 print(avg(5))
13 print(avg(7))
14 print(avg(6))
```





Décorateur

Ajouter un comportement à une fonction

- Fonction `decorate` qui prend **une fonction en paramètre**

Ajout d'un comportement avant et après l'appel d'une fonction

```
1 def decorate(f):
2     def wrapper():
3         print('Avant appel')
4         f()
5         print('Après appel')
6     return wrapper
7
8 def sayhello():
9     print('Hello!')
10
11 g = decorate(sayhello)
12 g()
```



```
Avant appel
Hello!
Après appel
```

Décoration

- Une **décoration** est ajoutée explicitement à une fonction

Se note avec @ suivi du nom de la fonction décorante

- Un **sucré syntaxique** est une notation simplificatrice

sayhello = decorate(sayhello)

```
1 def decorate(f):
2     def wrapper():
3         print('Avant appel')
4         f()
5         print('Après appel')
6     return wrapper
7
8 @decorate
9 def sayhello():
10    print('Hello!')
11
12 sayhello()
```



Chronométrer le temps d'exécution

- Annotation `@timeit` pour afficher le **temps d'exécution**

Mesure du temps avant et après exécution et calcul différence

- **Paramètre `*args`** permet d'en recevoir un nombre indéterminé

args est un tuple contenant les paramètres

```
1 import time
2
3 def timeit(f):
4     def wrapper(*args): 
5         t1 = time.time()
6         result = f(*args)
7         print('Executed in {:.2f} s'.format(time.time() - t1))
8         return result
9     return wrapper 
```

Vérification de l'authentification

- Annotation `@login_required` pour vérifier l'authentification

Vérification d'une variable globale user

```
1 from bottle import route, redirect, run
2
3 user = None
4
5 def login_required(f):
6     def wrapper(*args):
7         if user == None:
8             redirect('/login')
9         return f(*args)
10    return wrapper
11
12 @route('/myaccount')
13 @login_required
14 def myaccount():
15     return 'OK...'
16
17 run(host='localhost', port=8080)
```



Paramètre de décorateur (1)

- Un décorateur peut prendre des **paramètres**

Ils sont passés à la fonction décorante

```
1 def checktypes(*types):
2     def decorator(f):
3         def wrapper(*args):
4             return f(*args)
5         return wrapper
6     return decorator
7
8 @checktypes(int)
9 def compute(n):
10    return n ** 2
11
12 print(compute(9))
```



Paramètre de décorateur (2)

- Deux niveaux d'imbrication de fonctions

checktypes renvoie une fonction qui en prend une en paramètre

```
1 def checktypes(*types):
2     def decorator(f):
3         def wrapper(*args):
4             return f(*args)
5         return wrapper
6     return decorator
7
8 def compute(n):
9     return n ** 2
10
11 f1 = checktypes(int)
12 f2 = f1(compute)
13 print(f2(9))
```



Vérification des types

- Utilisation d'**assertions** pour vérifier les types des paramètres
 - Même nombre de paramètres que de types
 - Chaque paramètre possède le bon type

```
1 def checktypes(*types):  
2     def decorator(f):  
3         def wrapper(*args):  
4             assert len(types) == len(args), 'wrong number of types'  
5             for (a, t) in zip(args, types):  
6                 assert isinstance(a, t), 'arg {} does not match {}'.  
7                 .format(a, t)  
8             return f(*args)  
9         return wrapper  
return decorator
```





Générateur

Itérateur

- Parcourir une collection avec un **itérateur**

Toutes les collections de Python sont itérables



- Utilisé en interne dans plusieurs situations

- Avec la boucle `for`
- Constructions de collections avec `list`, `set`, `dict`...
- Définition de liste, ensemble et dictionnaire par compréhension
- Déballage de tuple
- Déballage du paramètre `*` des fonctions

Définition d'un itérateur (1)

- Redéfinition de la **méthode `__getitem__`**



Elle reçoit un indice en paramètre et renvoie la valeur associée

- Redéfinition de `__len__` pour définir la taille

```
1 import re
2
3 RE_WORDS = re.compile(r'\w+')
4
5 class Sentence:
6     def __init__(self, text):
7         self.__text = text
8         self.__words = RE_WORDS.findall(text)
9
10    def __getitem__(self, i):
11        return self.__words[i]
12
13    def __len__(self):
14        return len(self.__words)
```



Définition d'un itérateur (2)

- Utilisation possible de la **boucle for**

Parcours transparent de la nouvelle collection ainsi définie



```
1 s = Sentence("Bonjour, c'est moi.")  
2 print(list(s))  
3 for word in s:  
4     print(word)
```

```
[ 'Bonjour' , 'c' , 'est' , 'moi' ]  
Bonjour  
c  
est  
moi
```

Itérateur vs Itérable

- Une collection est **itérable** si on sait la parcourir
Il suffit d'avoir redéfini la méthode `__getitem__`
- Python utilise un **itérateur** pour parcourir une collection
Utilisation des fonctions `iter` et `next`

```
1 data = [1, 8, -2, 4]
2 it = iter(data)
3 while True:
4     try:
5         print(next(it), end=' ')
6     except StopIteration:
7         del it
8         break
```

```
1 8 -2 4
```

Définir un itérateur (1)

- Définition des méthodes `__iter__` et `__next__`

Objet représentant l'itérateur et élément suivant

```
1 class Sentence:  
2     def __init__(self, text):  
3         self.__text = text  
4         self.__words = RE_WORDS.findall(text)  
5  
6     def __iter__(self):  
7         self.__next = 0  
8         return self  
9  
10    def __next__(self):  
11        try:  
12            word = self.__words[self.__next]  
13        except IndexError:  
14            raise StopIteration()  
15        self.__next += 1  
16        return word
```



Définir un itérateur (2)

- Autoriser l'existence de **plusieurs itérateurs** avec un objet

Il faut définir une nouvelle classe représentant l'itérateur

```
1 class SentenceIterator:  
2     def __init__(self, words):  
3         self.__words = words  
4         self.__index = 0  
5  
6     def __iter__(self):  
7         return self  
8  
9     def __next__(self):  
10        try:  
11            word = self.__words[self.__index]  
12        except IndexError:  
13            raise StopIteration()  
14        self.__index += 1  
15        return word
```

Définir un itérateur (3)

- Renvoi d'une instance de la **classe représentant l'itérateur**

Autorise plusieurs itérateurs dans des états différents

```
1 class Sentence:  
2     def __init__(self, text):  
3         self.__text = text  
4         self.__words = RE_WORDS.findall(text)  
5  
6     def __iter__(self):  
7         return SentenceIterator(self.__words)
```

**Un itérateur est itérable,
mais un itérable n'est pas un itérateur**

Générateur (1)

- Un **générateur** crée des données au besoin

Utile lorsqu'on ne peut pas tout pré-générer en mémoire

- Le **mot réservé yield** produit la donnée suivante

Lorsque yield est présent, on a un générateur

```
1 def data():
2     yield 1
3     yield 2
4     yield 3
5
6 d = data()
7 print(d)
```



```
<generator object data at 0x10954f830>
```

Générateur (2)

- Parcours à l'aide d'une boucle `for`

Ou en utilisant la fonction `next`

- Les données ne sont **produites** que lorsque demandées

```
1 d = data()  
2 for elem in d:  
3     print(elem)
```

```
1 d = data()  
2 while True:  
3     try:  
4         print(next(d))  
5     except StopIteration:  
6         break
```



```
1  
2  
3
```

Génération fainéante

■ Génération des factorielles de tous les nombres naturels

Utilisation de yield pour les générer à la demande

```
1 def fact():
2     n = 0
3     result = 1
4     while True:
5         yield (n, result)
6         n += 1
7         result *= n
8
9 genfact = fact()
10 for i in range(5):
11     print(next(genfact), end=' ')
```



```
(0, 1) (1, 1) (2, 2) (3, 6) (4, 24)
```

Définition par compréhension fainéante

- Génération des factorielles de tous les nombres naturels

Utilisation de yield pour les générer à la demande

```
1 squaredfact = (x[1] ** 2 for x in fact())
2 for i in range(3):
3     print(next(squaredfact), end=' ')
```



```
1 1 4
```

Livres de référence



ISBN

978-1-491-94600-8



ISBN

978-0-262-51087-5

Crédits

- Photos des livres depuis Amazon
- <https://www.flickr.com/photos/vestman/4908148942>
- <https://www.flickr.com/photos/wiredforsound23/8919238583>
- <https://www.flickr.com/photos/tsayles/2170044559>