Le paradigme Fonctionnel en Python

Memento

I. Lambda-expressions : syntaxe

En Python, les fonctions sont des objets comme les autres. Elles peuvent donc être paramètres d'autres fonctions. Quand on a besoin d'une fonction pour un usage unique, on peut définir une fonction anonyme appelée **lambda-expression**.

```
f = lambda x: x+1
g = lambda x, y: x + y
h = lambda: 4
```

II. Évaluation paresseuse

On appelle **évaluation paresseuse** le fait qu'un calcul ne soit déclenché que lorsqu'il est vraiment nécessaire. Ainsi, en Python, l'instruction `intervalle = range(1000000000000)` ne calcule pas réellement l'intervalle en question. Les valeurs seront calculées les une après les autres, uniquement lorqu'on en aura besoin.

III. Fonctions de base

Ces fonctions, paresseuses, sont dans module *builtins*, et donc accessibles sans nécessiter d'importer de module.

```
map : map(a -> b, iter([a]) -> iter[b]
```

Exemple :

```
>>> map(lambda x: x**2, [2, 4, 5])
<map object at 0x7f2d7bd3ac20>
>>> list(map(lambda x: x**2, [2, 4, 5])
[4,16, 25]
```

Applique une fonction à tous les éléments d'un itéable, e et renvoie l'itérable des résultats ;

```
filter : filter(a -> Bool, iter[a]) -> iter[a]
```

Renvoie un itérable constitué des élements de l'itérables passés en 2è paramètre pour lesquels le prédicat passé en premier paramètre est vérifié.

Exemple :

```
>>> list(filter(lambda mot: mot.count("o") > 0, ["bonjour", "tout", "le",
"monde"]))
["bonjour", "tout", "monde"]
```

IV. Fonction de *functools*

```
reduce : reduce(b -> a -> b, iter[a], b) -> b
```

"Replie" un itérable dans un accumulateur (initialisé grâce au 3è paramètre) grâce à la fonction d'accumulation passée en 1er paramètre.

Le 3è paramètre est optionnel ; s'il est absent, l'accumumaleur est initalisé avec le premier élément de l'iterable, et au commence au 2è élément.

Exemple :

```
>>> reduce(lambda acc,elem : acc + elem ** 2, [1, 2, 3])
14
```


V. Fonctions de *itertools*

Toutes ces fontions sont paresseuses.

```
takewhile: takewhile(a -> bool, iter[a]) -> iter[ab]
```

Génère un itérable de tous les éléments du début de l'itérable passé en paramètre vérifiant le prédicat. S'arrête dès qu'un élément ne vérifie pas le prédicat.

Exemple :

```
list(takewhile(lambda mot: mot.count("o") > 0, ["bonjour", "tout", "le",
"monde"]))
["bonjour", "tout"]
```

```
dropwhile: dropwhile(a -> Bool, iter[a]) -> iter[ab]
```

Saute tous les éléments au début de l'itérable passé en paramètre vérifiant le prédicat. Renvoie l'itérable des éléments qui suivent (qu'ils vérifient ou non le prédicat).

Exemple :

```
list(dropwhile(lambda mot: mot.count("o") > 0, ["bonjour", "tout", "le",
"monde"]))
["le", "monde"]
```

```
count: count(int, int) -> iter[int]
```

Génère un itérable "infini" de nombres, en commençant au 1er paramètre (optionnel) et par pas du 2è paramètre (optionnel).

Exemple :

```
list(dropwhile(lambda x: x < 5, takewhile(lambda x: x < 10, count())))
```

Remarque : il existe de nombreuses autres fonctions dans le module *itertools* :

https://docs.python.org/3/library/itertools.html

VI. Rappel sur la notion d'itérable

Un **itérable** est une structure de données que l'on peut parcourir grâce à un *itérateur*. En python, un objet est itérable si sa classe définie une méthode d'instance `__iter__` qui renvoie un itérateur sur l'objet. La fonction `iter` permet de créer un itérateur à partir d'un objet itérable en appelant cette méthode.

Un **itérateur** est un objet à usage unique qui permet de parcourir les éléments de l'itérable à partir duquel il a été créé. La classe d'un térateur définit une méthode d'instance __next__ qui renvoie l'objet courant de la structure de données et passe au suivant. S'il n'y a plus d'objet courant, elle lève une exception StopIteration. La fonction next() permet sur un itérateur permet d'appeler cette méthode.

En Python, les listes, les dictionnaires, les ensembles, les intervalles sont des itérables.

La boucle for de Python repose sur ces notions d'itérable et d'itérateur. Ainsi, la boucle suivante :

```
for i in range(5):
  print(i)
```

est en fait du sucre syntaxique pour le code suivant :

```
intervalle = range(5)
iterateur = iter(intervalle)
try:
  while True:
    element = next(iterateur)
    print(element)
except StopIteration:
  pass
```