seance5_approche_fonctionnelle_enonce

July 2, 2023

1 Données, approches fonctionnelles - énoncé

L'approche fonctionnelle est une façon de traiter les données en ne conservant qu'une petite partie en mémoire. D'une manière générale, cela s'applique à tous les calculs qu'on peut faire avec le langage SQL. Le notebook utilisera des données issues d'une table de mortalité extraite de table de mortalité de 1960 à 2010 (*le lien est cassé car data-publica ne fournit plus ces données, le notebook récupère une copie*) qu'on récupère à l'aide de la fonction table_mortalite_euro_stat.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('ggplot')
import pyensae
from pyquickhelper.helpgen import NbImage
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

```
Populating the interactive namespace from numpy and matplotlib
```

[1]:
```
<IPython.core.display.HTML object>
```

[2]:
```
from actuariat_python.data import table_mortalite_euro_stat
table_mortalite_euro_stat()
import pandas
df = pandas.read_csv("mortalite.txt", sep="\t", encoding="utf8", low_memory=False)
df.head()
```

[2]:

	annee	valeur	age	age_num	indicateur	genre	pays
0	2012	0.00000	Y01	1.0	DEATHRATE	F	AD
1	2014	0.00042	Y01	1.0	DEATHRATE	F	AL
2	2009	0.00080	Y01	1.0	DEATHRATE	F	AM
3	2008	0.00067	Y01	1.0	DEATHRATE	F	AM
4	2007	0.00052	Y01	1.0	DEATHRATE	F	AM

1.0.1 Itérateur, Générateur

itérateur La notion d'itérateur est incournable dans ce genre d'approche fonctionnelle. Un itérateur parcourt les éléments d'un ensemble. C'est le cas de la fonction range.

[3]:
```
it = iter([0,1,2,3,4,5,6,7,8])
print(it, type(it))
```

```
<list_iterator object at 0x000001A50784C358> <class 'list_iterator'>
```

Il faut le dissocier d'une liste qui est un conteneur.

[4]:
```
[0,1,2,3,4,5,6,7,8]
```

[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8]

Pour s'en convaincre, on compare la taille d'un itérateur avec celui d'une liste : la taille de l'itérateur ne change pas quelque soit la liste, la taille de la liste croît avec le nombre d'éléments qu'elle contient.

[5]:
```
import sys
print(sys.getsizeof(iter([0,1,2,3,4,5,6,7,8])))
print(sys.getsizeof(iter([0,1,2,3,4,5,6,7,8,9,10,11,12,13,14])))
print(sys.getsizeof([0,1,2,3,4,5,6,7,8]))
print(sys.getsizeof([0,1,2,3,4,5,6,7,8,9,10,11,12,13,14]))
```

```
56
56
136
184
```

L'itérateur ne sait faire qu'une chose : passer à l'élément suivant et lancer une exception StopIteration lorsqu'il arrive à la fin.

[6]:
```
it = iter([0,1,2,3,4,5,6,7,8])
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

```
0
1
2
3
4
5
6
7
8
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-6-3260960c0f0b> in <module>()
      9 print(next(it))
     10 print(next(it))
---> 11 print(next(it))

StopIteration:
```

générateur Un générateur se comporte comme un itérateur, il retourne des éléments les uns à la suite des autres que ces éléments soit dans un container ou pas.

[7]:
```
def genere_nombre_pair(n):
    for i in range(0,n):
        yield 2*i

genere_nombre_pair(5)
```

[7]:
```
<generator object genere_nombre_pair at 0x000001A507A81FC0>
```

Appelé comme suit, un générateur ne fait rien. On s'en convaint en insérant une instruction `print` dans la fonction :

[8]:
```
def genere_nombre_pair(n):
    for i in range(0,n):
        print("je passe par là", i, n)
        yield 2*i

genere_nombre_pair(5)
```

[8]:
```
<generator object genere_nombre_pair at 0x000001A507A81F68>
```

Mais si on construit une liste avec tout ces nombres, on vérifie que la fonction `genere_nombre_pair` est bien executée :

[9]:
```
list(genere_nombre_pair(5))
```

```
je passe par là 0 5
je passe par là 1 5
je passe par là 2 5
je passe par là 3 5
je passe par là 4 5
```

[9]:
```
[0, 2, 4, 6, 8]
```

L'instruction `next` fonctionne de la même façon :

[10]:
```
def genere_nombre_pair(n):
    for i in range(0,n):
        yield 2*i

it = genere_nombre_pair(5)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

```
0
2
4
6
8
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
```

```
<ipython-input-10-cb3bdd50dd95> in <module>()
      9 print(next(it))
     10 print(next(it))
---> 11 print(next(it))

StopIteration:
```

Le moyen le plus simple de parcourir les éléments retournés par un itérateur ou un générateur est une boucle `for` :

[11]:
```
it = genere_nombre_pair(5)
for nombre in it:
    print(nombre)
```

```
0
2
4
6
8
```

On peut combiner les générateurs :

[12]:
```
def genere_nombre_pair(n):
    for i in range(0,n):
        print("pair", i)
        yield 2*i

def genere_multiple_six(n):
    for pair in genere_nombre_pair(n):
        print("six", pair)
        yield 3*pair

print(genere_multiple_six)
```

```
<function genere_multiple_six at 0x000001A507A94EA0>
```

[13]:
```
for i in genere_multiple_six(3):
    print(i)
```

```
pair 0
six 0
0
pair 1
six 2
6
pair 2
six 4
12
```

intérêt

- Les itérateurs et les générateurs sont des fonctions qui parcourent des ensembles d'éléments ou donne cette illusion.

- Ils ne servent qu'à passer à l'élément suivant.
- Ils ne le font que si on le demande explicitement avec une boucle `for` par exemple. C'est pour cela qu'on parle d'évaluation paresseuse ou **lazy evaluation**.
- On peut combiner les itérateurs / générateurs.

Il faut voir les itérateurs et générateurs comme des flux, une ou plusieurs entrées d'éléments, une sortie d'éléments, rien ne se passe tant qu'on n'envoie pas de l'eau pour faire tourner la roue.

lambda fonction Une fonction lambda est une fonction plus courte d'écrire des fonctions très simples.

[14]:
```
def addition(x, y):
    return x + y
addition(1, 3)
```

[14]: 4

[15]:
```
additionl = lambda x,y : x+y
additionl(1, 3)
```

[15]: 4

1.0.2 Exercice 1 : application aux grandes bases de données

Imaginons qu'on a une base de données de 10 milliards de lignes. On doit lui appliquer deux traitements : `f1`, `f2`. On a deux options possibles :

- Appliquer la fonction `f1` sur tous les éléments, puis appliquer `f2` sur tous les éléments transformés par `f1`.
- Application la combinaison des générateurs `f1`, `f2` sur chaque ligne de la base de données.

Que se passe-t-il si on a fait une erreur d'implémentation dans la fonction `f2` ?

[16]:
```

```

1.0.3 Map/Reduce, approche fonctionnelle avec cytoolz

On a vu les fonctions iter et next mais on ne les utilise quasiment jamais. La programmation fonctionnelle consiste le plus souvent à combiner des itérateurs et générateurs pour ne les utiliser qu'au sein d'une boucle. C'est cette boucle qui appelle implicitement les deux fonctions iter et next.
La combinaison d'itérateurs fait sans cesse appel aux mêmes schémas logiques. Python implémente quelques schémas qu'on complète par un module tel que cytoolz. Les deux modules toolz et cytoolz sont deux implémentations du même ensemble de fonctions décrit par la documentation : pytoolz. toolz est une implémentation purement Python. cytoolz s'appuie sur le langage C++, elle est plus rapide.
Par défault, les éléments entrent et sortent dans le même ordre. La liste qui suit n'est pas exhaustive (voir itertoolz).
schémas simples:

- filter : sélectionner des éléments, n qui entrent, $< n$ qui sortent.
- map : transformer les éléments, n qui entrent, n qui sortent.
- take : prendre les k premiers éléments, n qui entrent, $k <= n$ qui sortent.
- drop : passer les k premiers éléments, n qui entrent, $n - k$ qui sortent.
- sorted : tri les éléments, n qui entrent, n qui sortent dans un ordre différent.
- reduce : aggréger (au sens de sommer) les éléments, n qui entrent, 1 qui sort.
- concat : fusionner deux séquences d'éléments définies par deux itérateurs, n et m qui entrent, $n + m$ qui sortent.

schémas complexes
Certains schémas sont la combinaison de schémas simples mais il est plus efficace d'utiliser la version combinée.

- join : associe deux séquences, n et m qui entrent, au pire nm qui sortent.
- groupby : classe les éléments, n qui entrent, $p <= n$ groupes d'éléments qui sortent.
- reduceby : combinaison (*groupby*, *reduce*), n qui entrent, $p <= n$ qui sortent.

schéma qui retourne un seul élément

- all : vrai si tous les éléments sont vrais.
- any : vrai si un éléments est vrai.
- first : premier élément qui entre.
- last : dernier élément qui sort.
- min, max, sum, len…

schéma qui aggrège

- add : utilisé avec la fonction *reduce* pour aggréger les éléments et n'en retourner qu'un.

API PyToolz décrit l'ensemble des fonctions disponibles.

1.0.4 Exercice 2 : cytoolz

La note d'un candidat à un concours de patinage artistique fait la moyenne de trois moyennes parmi cinq, les deux extrêmes n'étant pas prises en compte. Il faut calculer cette somme pour un ensemble de candidats avec cytoolz.

[17]:
```
notes = [dict(nom="A", juge=1, note=8),
         dict(nom="A", juge=2, note=9),
         dict(nom="A", juge=3, note=7),
         dict(nom="A", juge=4, note=4),
         dict(nom="A", juge=5, note=5),
         dict(nom="B", juge=1, note=7),
         dict(nom="B", juge=2, note=4),
         dict(nom="B", juge=3, note=7),
         dict(nom="B", juge=4, note=9),
         dict(nom="B", juge=1, note=10),
         dict(nom="C", juge=2, note=0),
         dict(nom="C", juge=3, note=10),
         dict(nom="C", juge=4, note=8),
         dict(nom="C", juge=5, note=8),
         dict(nom="C", juge=5, note=8),
         ]

import pandas
pandas.DataFrame(notes)
```

[17]:

	juge	nom	note
0	1	A	8
1	2	A	9
2	3	A	7
3	4	A	4
4	5	A	5
5	1	B	7
6	2	B	4

```
7      3   B     7
8      4   B     9
9      1   B    10
10     2   C     0
11     3   C    10
12     4   C     8
13     5   C     8
14     5   C     8
```

[18]:
```python
import cytoolz.itertoolz as itz
import cytoolz.dicttoolz as dtz
from functools import reduce
from operator import add
```

[19]:
```python

```

1.0.5 Blaze, odo : interfaces communes

Blaze fournit une interface commune, proche de celle des Dataframe, pour de nombreux modules comme bcolz… odo propose des outils de conversions dans de nombreux formats.

- Pandas to Blaze

Ils sont présentés dans un autre notebook. On reproduit ce qui se fait une une ligne avec odo.

[20]:
```python
df.to_csv("mortalite_compresse.csv", index=False)
```

[21]:
```python
from pyquickhelper.filehelper import gzip_files
gzip_files("mortalite_compresse.csv.gz", ["mortalite_compresse.csv"], encoding="utf-8")
```

1.0.6 Parallélisation avec dask

- dask

dask propose de paralléliser les opérations usuelles qu'on applique à un dataframe.
L'opération suivante est très rapide, signifiant que *dask* attend de savoir quoi faire avant de charger les données :

[22]:
```python
import dask.dataframe as dd
fd = dd.read_csv('mortalite_compresse*.csv.gz', compression='gzip', blocksize=None)
#fd = dd.read_csv('mortalite_compresse.csv', blocksize=None)
```

Extraire les premières lignes prend très peu de temps car *dask* ne décompresse que le début :

[23]:
```python
fd.head()
```

[23]:

	annee	valeur	age	age_num	indicateur	genre	pays
0	2012	0.00000	Y01	1.0	DEATHRATE	F	AD
1	2014	0.00042	Y01	1.0	DEATHRATE	F	AL
2	2009	0.00080	Y01	1.0	DEATHRATE	F	AM
3	2008	0.00067	Y01	1.0	DEATHRATE	F	AM
4	2007	0.00052	Y01	1.0	DEATHRATE	F	AM

[24]:
```python
fd.npartitions
```

[24]: 1

[25]:
```
fd.divisions
```

[25]: (None, None)

[26]:
```
s = fd.sample(frac=0.01)
```

[27]:
```
s.head()
```

[27]:

	annee	valeur	age	age_num	indicateur	genre	pays
2514555	2012	1.061787e+06	NaN	NaN	TOTPYLIVED	F	AD
1812522	2006	7.691800e+04	Y61	61.0	PYLIVED	M	AM
2352885	2008	7.170500e+04	Y68	68.0	SURVIVORS	T	RO
1018976	2013	1.162000e-02	Y62	62.0	PROBDEATH	M	MT
1960726	1999	7.475100e+04	Y70	70.0	PYLIVED	T	IE

[28]:
```
life = fd[fd.indicateur=='LIFEXP']
life
```

[28]: dd.DataFrame<getitem…, npartitions=1>

[29]:
```
life.head()
```

[29]:

	annee	valeur	age	age_num	indicateur	genre	pays
398874	2012	91.3	Y01	1.0	LIFEXP	F	AD
398875	2014	79.9	Y01	1.0	LIFEXP	F	AL
398876	2009	76.5	Y01	1.0	LIFEXP	F	AM
398877	2008	76.4	Y01	1.0	LIFEXP	F	AM
398878	2007	76.5	Y01	1.0	LIFEXP	F	AM

[30]: