dataframe_matrix_speed

August 5, 2022

1 2A.i - Mesures de vitesse sur les dataframes

Lire un dataframe avec un itérateur quand on ne connaît pas sa taille, lire un array avec un itérateur.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

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

1.1 Création d'un dataframe à partir d'un itérateur

On cherche à créer un dataframe à partir d'un ensemble de lignes dont on ne connaît pas le nombre au moment où on créé le dataframe car on les reçoit sous la forme d'un itérateur ou un générateur.

[2]:
```
import random
def enumerate_row(nb=10000, n=10):
    for i in range(nb):
        # on retourne un tuple, les données sont
        # plus souvent recopiées car le type est immuable
        yield tuple(random.random() for k in range(n))
        # on retourne une liste, ces listes ne sont pas
        # recopiées en général, seule la liste qui les tient
        # l'est
        # yield list(random.random() for k in range(n))

list(enumerate_row(2))
```

[2]:
```
[(0.48407052966425546,
  0.804134458177794,
  0.6429087806017728,
  0.15971606067266708,
  0.24228260345690167,
  0.23378976056237966,
  0.689702986896856,
  0.6968197701594067,
  0.644936143902689,
  0.17127561953185932),
 (0.37058334468191145,
  0.689477712320184,
  0.02131118805602783,
  0.772192897946077,
  0.7048866020231711,
```

```
  0.012595878354861978,
  0.23269748071952856,
  0.5331829925096959,
  0.29721238636977587,
  0.884680186273301)]
```

[3]:
```
import pandas
nb, n = 10, 10
df = pandas.DataFrame(enumerate_row(nb=nb, n=n), columns=["c%d" % i for i in range(n)])
df.head()
```

[3]:

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
0	0.441234	0.424149	0.672914	0.890296	0.368575	0.477571	0.798417	0.694960	0.439395	0.910571
1	0.533190	0.654924	0.316732	0.218678	0.509369	0.329770	0.384639	0.674780	0.744674	0.776442
2	0.863879	0.710917	0.387745	0.520868	0.424982	0.864928	0.639818	0.953759	0.855713	0.783866
3	0.794047	0.511736	0.857161	0.625803	0.352731	0.993898	0.195672	0.476582	0.293988	0.796978
4	0.483821	0.119623	0.647837	0.215354	0.144840	0.119718	0.159295	0.184282	0.919450	0.612818

[4]:
```
nb, n =100000, 10
```

On compare plusieurs constructions :

[5]:
```
print(nb,n)
%timeit pandas.DataFrame(enumerate_row(nb=nb,n=n), columns=["c%d" % i for i in␣
  ↪range(n)])
```

```
100000 10
1 loops, best of 3: 539 ms per loop
```

[6]:
```
print(nb,n)
%timeit pandas.DataFrame(list(enumerate_row(nb=nb,n=n)), columns=["c%d" % i for i in␣
  ↪range(n)])
```

```
100000 10
1 loops, best of 3: 531 ms per loop
```

On décompose :

[7]:
```
def cache():
    return list(enumerate_row(nb=nb,n=n))
print(nb,n)
%timeit cache()
```

```
100000 10
1 loops, best of 3: 437 ms per loop
```

[8]:
```
print(nb,n)
l = list(enumerate_row(nb=nb,n=n))
%timeit pandas.DataFrame(l, columns=["c%d" % i for i in range(n)])
```

```
100000 10
10 loops, best of 3: 118 ms per loop
```

D'après ces temps, pandas convertit probablement l'itérateur en liste. On essaye de créer le dataframe vide, puis avec la méthode from_records.

[9]:
```
%timeit pandas.DataFrame(columns=["c%d" % i for i in range(n)], index=range(n))
```

```
The slowest run took 4.15 times longer than the fastest. This could mean that an
intermediate result is being cached
1000 loops, best of 3: 1.58 ms per loop
```

[10]:
```
def create_df3():
    return pandas.DataFrame.from_records(enumerate_row(nb=nb,n=n),
                                         columns=["c%d" % i for i in range(n)])

print(nb,n)
%timeit create_df3()
```

```
100000 10
1 loops, best of 3: 508 ms per loop
```

1.2 Création d'un array à partir d'un itérateur

On cherche à créer un dataframe à partir d'un ensemble de lignes dont on ne connaît pas le nombre au moment où on créé le dataframe car on les reçoit sous la forme d'un itérateur ou un générateur. La documentation de la fonction numpy.fromiter est intéressante à ce sujet.

[11]:
```
def enumerate_row2(nb=10000, n=10):
    for i in range(nb):
        for k in range(n):
            yield random.random()

import numpy
nb,n = 100000, 10
# on précise la taille du tableau car cela évite à numpy d'agrandir le tableau
# au fur et à mesure, ceci ne fonctionne pas
print(nb,n)
m = numpy.fromiter(enumerate_row2(nb=nb,n=n), float, nb*n)
m.resize((nb,n))
m [:5,:]
```

```
100000 10
```

[11]:
```
array([[ 0.56015574,  0.87562798,  0.98440727,  0.09243148,  0.71725052,
         0.46137005,  0.34867008,  0.16251999,  0.81366002,  0.04497673],
       [ 0.61909677,  0.71484276,  0.81124602,  0.35659089,  0.39326758,
         0.65078866,  0.92445948,  0.32325873,  0.3898043 ,  0.92096777],
```

```
[ 0.88257268,  0.92949871,  0.85010483,  0.13049573,  0.30860387,
  0.60728701,  0.95734593,  0.01977126,  0.06968685,  0.98705452],
[ 0.84260967,  0.96877873,  0.8886382 ,  0.63529283,  0.99503151,
  0.95205265,  0.80948905,  0.08377304,  0.68015408,  0.74213396],
[ 0.77371233,  0.39757708,  0.45769021,  0.01445757,  0.4160269 ,
  0.2832024 ,  0.68086284,  0.4086078 ,  0.99526844,  0.15037316]])
```

[12]:
```
def create_array():
    m = numpy.fromiter(enumerate_row2(nb=nb,n=n), float, nb*n)
    m.resize((nb,n))
    return m
print(nb,n)
%timeit create_array()
```

```
100000 10
1 loops, best of 3: 255 ms per loop
```

[13]:
```
def create_array2():
    m = list(enumerate_row(nb=nb,n=n))
    ml = numpy.array(m, float)
    return ml
print(nb,n)
%timeit create_array2()
```

```
100000 10
1 loops, best of 3: 430 ms per loop
```

Et si on ne précise pas la taille du tableau créé avec la fonction `fromiter` :

[14]:
```
def create_array3():
    m = numpy.fromiter(enumerate_row2(nb=nb,n=n), float)
    m.resize((nb,n))
    return m
print(nb,n)
%timeit create_array3()
```

```
100000 10
1 loops, best of 3: 271 ms per loop
```

On retrouve des temps similaires que ceux obtenus avec une liste. En conclusion, pour créer un *array*, il vaut mieux :

- connaître la taille finale
- éviter de créer une liste

Pour finir, je recommande la lecture de Enhancing Performance qui étudie différent scénari avec cython, eval, numba.

[15]:
```
```