Applications

# of Data Science

The Pandasverse

Applications of Data Science - Class 6

Giora Simchoni

`gsimchoni@gmail.com and add #dsapps in subject`

Stat. and OR Department, TAU

2020-03-01

APPLICATIONS

OF DATA SCIENCE

`Numpy`: Your best friend

APPLICATIONS

OF DATA SCIENCE

Python was not made for Data Science

```
mean([1, 2, 3, 4, 5])
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: na
'mean' is not defined
##
## Detailed traceback:
## File "", line 1, in
```

Enter `Numpy` to the rescue:

```
import numpy as np

np.mean(np.array([1, 2, 3, 4, 5]))
```

```
## 3.0
```

```
np.array([1, 2, 3, 4, 5]).mean()
```

```
## 3.0
```

Numpy Arrays

Create with a list:

```
a = np.array([1, 2, 3])
print(type(a))
```

```
## <class 'numpy.ndarray'>
```

```
print(a.shape)
```

```
## (3,)
```

> ⚠ Index is zero based!

```
print(a[0])
```

```
## 1
```

Create a 2D array:

```
b = np.array([[1,2,3],[4,5,6]])
print(b)
```

```
## [[1 2 3]
##  [4 5 6]]
```

```
print(b.shape)
```

```
## (2, 3)
```

Many ways to create "typical" arrays:

```
# create an array of all zeros
# (the parameter is a tuple specifying the array shape)
a = np.zeros((2,2))

# create an array of all ones
b = np.ones((1,2))

# create a constant array
c = np.full((2,2), 7)

# create a 2x2 identity matrix
d = np.eye(2)

# create an array filled with random U(0, 1) values
e = np.random.random((2,2))

# create a sequence from 2 to 15, not including
np.arange(2, 15)

# create sequence of 11 numbers between 0 and 1 including
np.linspace(0, 1, 11)
```

And every array has a `reshape()` method:

```
np.arange(0.1, 1, step=0.1).reshape(3, 3)
```

```
## array([[0.1, 0.2, 0.3],
##        [0.4, 0.5, 0.6],
##        [0.7, 0.8, 0.9]])
```

Numpy Math

Elementwise multiplication:

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
print(x * 2)
```

```
## [[2. 4.]
##  [6. 8.]]
```

Elementwise sum:

```
print(x + y)
```

```
## [[ 6.  8.]
##  [10. 12.]]
```

Same:

```
print(np.add(x, y))
```

You get the idea:

```
print(x - y)
print(np.subtract(x, y))

print(x * y)
print(np.multiply(x, y))

print(x / y)
print(np.divide(x, y))

print(np.sqrt(x))
```

Vector/Matrix multiplication:

```
print(x.dot(y))
```

```
## [[19. 22.]
##  [43. 50.]]
```

```
print(np.dot(x, y))
```

```
## [[19. 22.]
##  [43. 50.]]
```

```
v = np.array([9,10])
w = np.array([11, 12])

print(v.dot(w))
```

```
## 219
```

```
print(np.dot(v, w))
```

```
## 219
```

Transpose

```
x = np.array([[1,2],[3,4]])
print(x.T)
```

```
## [[1 3]
##  [2 4]]
```

Sum, mean, std, median, quantile, min, max...:

```
print(np.sum(x))  # Compute sum of all elements
```

```
## 10
```

```
print(np.sum(x, axis=0))  # Compute sum of each column
```

```
## [4 6]
```

```
print(np.std([1,2,3])) # possible, in case you were wondering
```

```
## 0.816496580927726
```

`Numpy` Indexing and Slicing

Similar to R but there are some things worth noticing:

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
```

```
## [[ 1  2  3  4]
##  [ 5  6  7  8]
##  [ 9 10 11 12]]
```

```
# use slicing to pull out the subarray consisting of the first 2 r
# and columns 1 and 2; b of shape (2, 2)
b = a[:2, 1:3]
print(b)
```

```
## [[2 3]
##  [6 7]]
```

```
# a slice of an array is a view into the same data, so modifying i
# will modify the original array.
print(a[0, 1])
```

```
## 2
```

```
b[0, 0] = 77
print(a[0, 1])
```

```
## 77
```

Very convenient, R does not have these features without external packages:

```
# index "from last place"
a[-2:]
```

```
## array([[ 5,  6,  7,  8],
##        [ 9, 10, 11, 12]])
```

```
# reverse an array
a = np.arange(5)
print(a[::-1])
```

```
## [4 3 2 1 0]
```

Working with boolean masks like in R:

```
print(a[a > 2])
```

```
## [3 4]
```

```
print(a[np.where(a > 2)])
```

```
## [3 4]
```

```
print(a[np.argmin(a)])
```

```
## 0
```

`Scipy`: Scientific Computing and Stats

Applications of Data Science

Many modules, let's focus on:

- **`sparse`**: Sparse Matrices manipulation
- **`ndimage`**: Images manipulation (though see `scikit-image` and `opencv`)
- **`stats`**: Statistics (though see `statsmodels`)

sparse

```
from scipy.sparse import csr_matrix

row = np.array([0, 0, 1, 2, 2, 2])
col = np.array([0, 2, 2, 0, 1, 2])
data = np.array([1, 2, 3, 4, 5, 6])
sparse_a = csr_matrix((data, (row, col)), shape=(3, 3))

print(sparse_a.toarray())
```

```
## [[1 0 2]
##  [0 0 3]
##  [4 5 6]]
```

ndimage

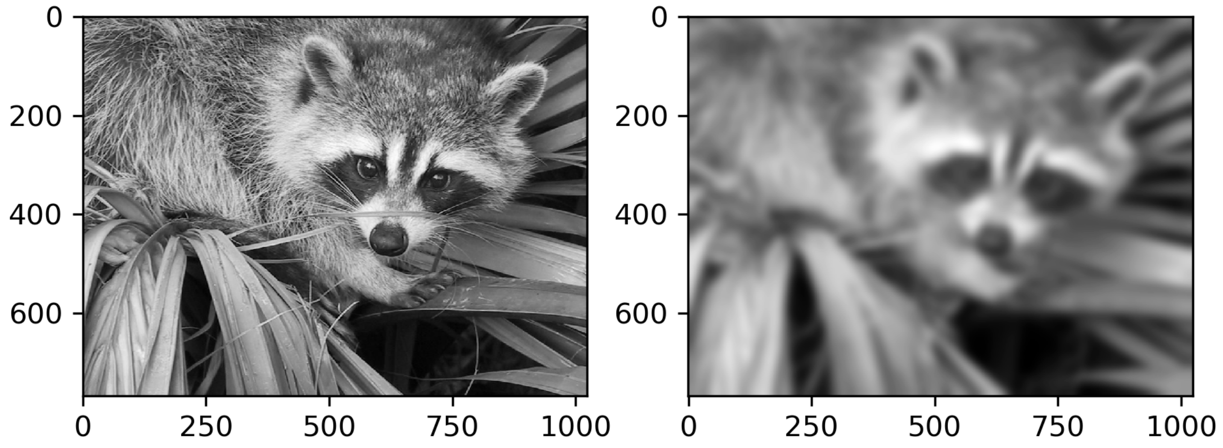
```
from scipy import ndimage
from scipy import misc
import matplotlib.pyplot as plt

face = misc.face(gray=True)
blurred_face = ndimage.gaussian_filter(face, sigma=10)

print(face.shape)
```

```
## (768, 1024)
```

```
print(face[:5, :5])
```

```
## [[114 130 145 147 147]
##  [ 83 104 123 130 134]
##  [ 68  88 109 116 120]
##  [ 78  94 109 116 121]
##  [ 99 109 119 128 138]]
```

```
plt.subplot(121)
plt.imshow(face, cmap=plt.cm.gray)
plt.subplot(122)
plt.imshow(blurred_face, cmap=plt.cm.gray)
plt.show()
```



stats

```
from scipy import stats

rvs1 = stats.norm.rvs(loc=5,scale=10,size=500)
rvs2 = stats.norm.rvs(loc=5,scale=10,size=500)

stats.ttest_ind(rvs1,rvs2)
```

```
## Ttest_indResult(statistic=-0.1672020280648639, pvalue=0.8672449606209668
```

`Pandas`: Data, Data, Data

Applications

Of Data Science

Series, DataFrames

A `Pandas Series` is a vector of data, a column.

```
import pandas as pd

s = pd.Series([1,3,5,np.nan,6,8])
print(s)
```

```
## 0    1.0
## 1    3.0
## 2    5.0
## 3    NaN
## 4    6.0
## 5    8.0
## dtype: float64
```

A `DataFrame` is a data table, always indexed.

Creating one from a random numpy 2D array (notice the index isn't specified, automatically becomes zero based counter):

```
df = pd.DataFrame(np.random.randn(6,4), columns = ['A', 'B', 'C',
print(df)
```

```
##          A         B         C         D
## 0  0.637024  0.328160 -0.893112  0.924013
## 1  1.659861 -0.531544  0.125794 -1.574434
## 2  0.029313 -0.394294 -0.708947 -0.046269
## 3 -0.719662  0.374894  2.028673 -0.751812
## 4 -0.969941  0.287350 -0.462109  0.971948
## 5  0.306526 -1.873495  0.965911 -1.425832
```

Creating a DataFrame from a very varied dictionary where each key is a column (also see `pd.from_dict()`).

```
df2 = pd.DataFrame({'A' : 1.,
                    'B' : pd.Timestamp('20130102'),
                    'C' : pd.Series(1, index = list(range(4)),
                      dtype = 'float32'),
                    'D' : np.array(np.arange(4), dtype = 'int32'),
                    'E' : pd.Categorical(
                      ["test", "train", "test", "train"]
                    ),
                    'F' : 'foo' })
print(df2)
```

```
##      A          B    C  D      E    F
## 0  1.0 2013-01-02  1.0  0   test  foo
## 1  1.0 2013-01-02  1.0  1  train  foo
## 2  1.0 2013-01-02  1.0  2   test  foo
## 3  1.0 2013-01-02  1.0  3  train  foo
```

read_csv()

```
okcupid = pd.read_csv("../data/okcupid.csv.zip")
```

```
okcupid.shape
```

```
## (59946, 31)
```

```
okcupid.columns
```

```
## Index(['age', 'body_type', 'diet', 'drinks', 'drugs', 'education', 'essa
##        'essay1', 'essay2', 'essay3', 'essay4', 'essay5', 'essay6', 'essa
##        'essay8', 'essay9', 'ethnicity', 'height', 'income', 'job',
##        'last_online', 'location', 'offspring', 'orientation', 'pets',
##        'religion', 'sex', 'sign', 'smokes', 'speaks', 'status'],
##       dtype='object')
```

info(), describe(), head() and tail()

```
okcupid.describe()
```

```
##                 age        height          income
## count  59946.000000  59943.000000    59946.000000
## mean      32.340290     68.295281    20033.222534
## std        9.452779      3.994803    97346.192104
## min       18.000000      1.000000       -1.000000
## 25%       26.000000     66.000000       -1.000000
## 50%       30.000000     68.000000       -1.000000
## 75%       37.000000     71.000000       -1.000000
## max      110.000000     95.000000  1000000.000000
```

```
okcupid.head(3)
```

```
##     age  ...     status
## 0    22  ...     single
## 1    35  ...     single
## 2    38  ...  available
##
## [3 rows x 31 columns]
```

Not `data.frame`, `DataFrame`

dplyr	pandas
`mutate`	`assign`
`select`	`filter`
`rename`	`rename`
`filter`	`query`
`arrange`	`sort_values`
`group_by`	`groupby`
`summarize`	`agg`

> Thare *are* Pandas dialects, don't go translating your pipes verbatim.

assign()

Add a column `height_cm`, the `height` in centimeters:

```
okcupid = okcupid.assign(height_cm = okcupid['height'] * 2.54)
okcupid = okcupid.assign(height_cm = lambda x: x.height * 2.54)
```

If you don't need a pipe just do:

```
okcupid['height_cm'] = okcupid['height'] * 2.54
```

`query()` and `filter()`

Query only women, filter only age and height:

```
okcupid \
  .query('sex == "f"') \
  .filter(['age', 'height']) \
  .head(5)
```

```
##      age  height
## 6     32    65.0
## 7     31    65.0
## 8     24    67.0
## 13    30    66.0
## 14    29    62.0
```

Again, without a pipe:

```
okcupid[okcupid['sex'] == "f"][['age', 'height']]
```

Same but income over 100K, and select all essay questions:

```
okcupid \
  .query('sex == "f" and income > 100000') \
  .filter(okcupid.columns[okcupid.columns.str.startswith('essay')]
```

```
##                                                   essay0  ...
## 48     i love it here, except when it's hotter than a...  ...   if you da
## 188    i'm silly. i'm analytical. i'm fond of short s...  ...   you want
## 301    welcome... i am one genuine, straight forward,...  ...
## 337    purebred cali girl! born and raised in nor cal...  ...   you are a
## 402    i wasn't like every other kid, you know, who d...  ...   you think
## ...                                                  ...  ...
## 59326  i am a forensic psychologist, mother, sister a...  ...
## 59395                                                NaN  ...
## 59789  i'm a fun loving woman, romantic, faithful, ea...  ...
## 59818  hello, i am usually pretty shy and sometimes a...  ...   you are pa
## 59819  this is a pretty good read. admittedly windy. ...  ...   you like
##
## [208 rows x 10 columns]
```

agg()

Find the average height of women

```
okcupid \
  .query('sex == "f"') \
  .filter(['height_cm']) \
  .agg('mean')
```

```
## height_cm    165.363837
## dtype: float64
```

Notice we got a `pd.Series`, the `Pandas` equivelent for a vector. We could use the `.values` attribute to pull the `Numpy` array behind the Series:

```
okcupid \
  .query('sex == "f"') \
  .filter(['height_cm']) \
  .agg('mean').values
```

```
## array([165.36383729])
```

groupby()

But why settle for women only?

```
okcupid \
  .groupby('sex')['height_cm'] \
  .agg('mean')
```

```
## sex
## f    165.363837
## m    178.926471
## Name: height_cm, dtype: float64
```

And you might want to consider `rename()`ing sex!

```
okcupid \
  .groupby('sex')['height_cm'] \
  .agg('mean') \
  .rename_axis(index = {'sex': 'gender'})
```

```
## gender
## f    165.363837
## m    178.926471
## Name: height_cm, dtype: float64
```

Group by multiple variables, get more summaries, arrange by descending average height:

```
okcupid \
  .groupby(['sex', 'status'])['height_cm'] \
  .agg(['mean', 'median', 'count']) \
  .sort_values('median', ascending=False)
```

```
##                              mean  median  count
## sex status
## m   available           179.445012  180.34   1209
##     married             179.454629  180.34    175
##     seeing someone      179.257926  177.80   1061
##     single              178.894660  177.80  33376
##     unknown             177.376667  176.53      6
## f   available           166.381616  166.37    656
##     married             165.871407  165.10    135
##     seeing someone      165.431745  165.10   1003
##     single              165.328643  165.10  22318
##     unknown             160.655000  158.75      4
```

Pro tip: `size()`

When all you want is, well, `size`:

```
okcupid.groupby('body_type').size()
```

```
## body_type
## a little extra      2629
## athletic           11819
## average            14652
## curvy               3924
## fit                12711
## full figured        1009
## jacked               421
## overweight           444
## rather not say       198
## skinny              1777
## thin                4711
## used up             355
## dtype: int64
```

`loc`, `iloc` and `at`

`loc` is for selection by name:

```
okcupid.loc[:3, ['sex', 'height_cm']]
```

```
##    sex  height_cm
## 0    m     190.50
## 1    m     177.80
## 2    m     172.72
## 3    m     180.34
```

The first element to `loc` slices the index by name. The reason that ":3" works is that our index is numeric. If it were for example `['a', 'b', 'c', ...]` it would not have worked.

`loc` can also accept boolean indexing:

```
okcupid.loc[okcupid['sex'] == 'm', 'height_cm']
```

`iloc` is for selection by integers on the index or column indices

```
okcupid.iloc[:3, 1:3]
```

```
##          body_type                   diet
## 0  a little extra  strictly anything
## 1         average       mostly other
## 2            thin           anything
```

This would have worked also if the index was `['a', 'b', 'c', ...]`.

Finally `at` is for accessing a specific value fast:

```
okcupid.at[1989, 'body_type']
```

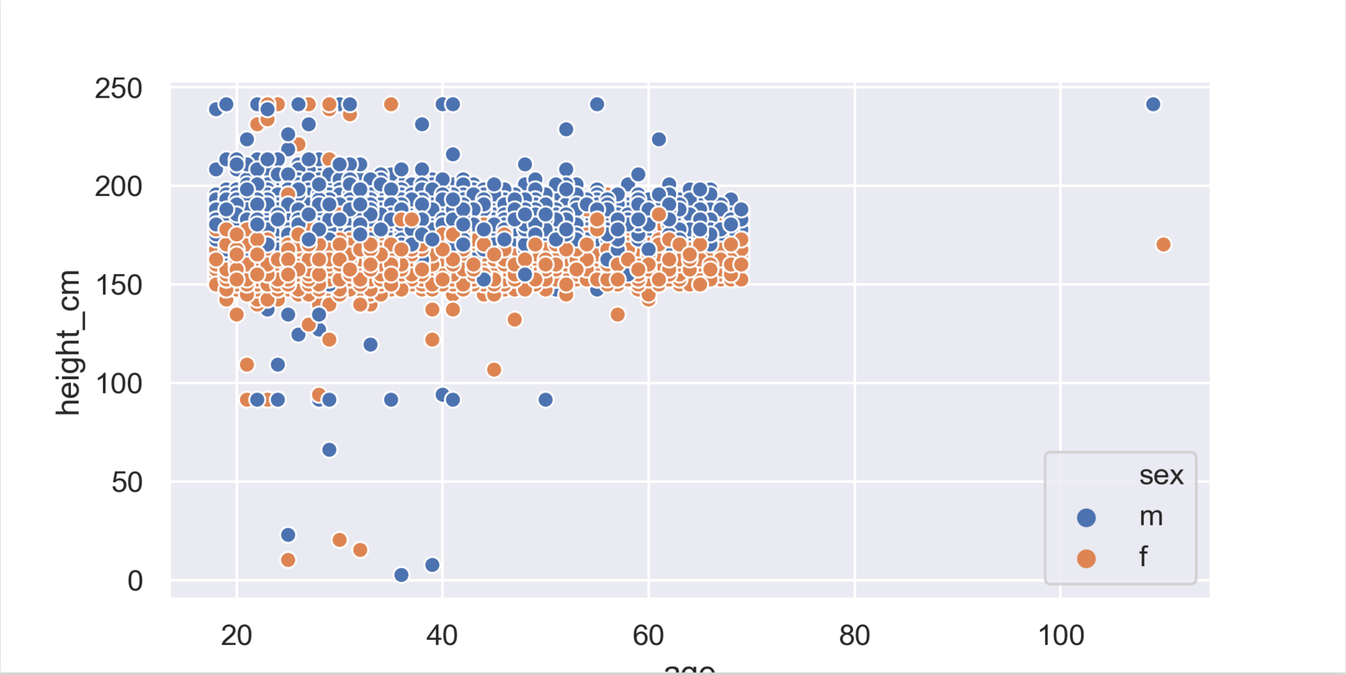
```
## 'average'
```

`seaborn`: Visualization

APPLICATIONS

OF DATA SCIENCE

```python
import matplotlib.pyplot as plt
import seaborn as sns

sns.set()
g = sns.scatterplot('age', 'height_cm', hue='sex', data = okcupid)
plt.show()
```



```python
g = sns.relplot('age', 'height_cm',
  hue = 'sex', kind = 'scatter', col='sex', data = okcupid)
plt.show()
```