



PRE-SESSION WK1

DAWSON, C.

Python for Data Science

Python Versions

```
python --version
```

```
sofija@sofija-VirtualBox:~$ python --version
Python 3.7.3
```

Data Types

The following list shows the seven most common Python data types:

- integer**: These are whole numbers, positive or negative (including 0). Example: `100`.
- float**: Floating-point numbers are real numbers, rational or irrational. In most cases, this means numbers with decimal fractions. Example: `123.45`.
- string**: Strings are sequences of characters, or text, enclosed in quotes. Example: `"any text"`.
- boolean**: Can be one of two values, true or false. Example: `True` or `False`.
- list**: An ordered sequence of elements. With a list, the order of the elements can be changed. Example: `[value1, value2, ...]`.
- tuple**: An unchangeable ordered sequence of elements. Example: `(value1, value2, ...)`.
- dictionary**: This is Python's mapping data type. Dictionaries map keys to values as a means of storing information. Example: `{key1:value1, key2:value2, ...}`.

Pandas

Python For Data Science Cheat Sheet

Pandas Basics

Learn Python for Data Science interactively at [www.DataCamp.com](https://www.datacamp.com)



Pandas

The Pandas library is built on NumPy and provides easy-to-use data structures and data analysis tools for the Python programming language.



Use the following import convention:
`>>> import pandas as pd`

Pandas Data Structures

Series

A one-dimensional labeled array capable of holding any data type

A	3
B	-5
C	7
D	4

Index

```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

DataFrame

Columns

	Country	Capital	Population
1	Belgium	Brussels	11190846
2	India	New Delhi	1303171035
3	Brazil	Brasilia	207847528

Index

A two-dimensional labeled data structure with columns of potentially different types

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
            'Capital': ['Brussels', 'New Delhi', 'Brasilia'],
            'Population': [11190846, 1303171035, 207847528]}

>>> df = pd.DataFrame(data,
                      columns=['Country', 'Capital', 'Population'])
```

I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> pd.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> pd.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')

Read multiple sheets from the same file
>>> xls = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xls, 'Sheet1')
```

Asking For Help

```
>>> help(pd.Series.loc)
```

Selection

Also see NumPy Arrays

Getting

```
>>> s['b']
-5
```

Get one element

```
>>> df[1:]
   Country  Capital  Population
1  India    New Delhi  1303171035
2  Brazil   Brasilia  207847528
```

Get subset of a DataFrame

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc[0, 0]
'Belgium'
>>> df.iat[0, 0]
'Belgium'
```

Select single value by row & column

By Label

```
>>> df.loc[0, ['Country']]
'Belgium'
>>> df.at[0, ['Country']]
'Belgium'
```

Select single value by row & column labels

By Label/Position

```
>>> df.ix[2]
Country    Brazil
Capital    Brasilia
Population  207847528
```

Select single row of subset of rows

```
>>> df.ix[:, 'Capital']
0    Brussels
1    New Delhi
2    Brasilia
```

Select a single column of subset of columns

```
>>> df.ix[1, 'Capital']
'New Delhi'
```

Select rows and columns

Boolean Indexing

```
>>> s[(s > 1)]
>>> s[(s < -1) | (s > 2)]
>>> df[df['Population'] > 1200000000]
```

Series s where value is not > 1
s where value is < -1 or > 2
Use filter to adjust DataFrame

Setting

```
>>> s['a'] = 6
```

Set index a of Series s to 6

Dropping

```
>>> s.drop(['a', 'c'])
>>> df.drop('Country', axis=1)
>>> df.drop('Country', axis=0)
```

Drop values from rows (axis=0)
Drop values from columns (axis=1)

Sort & Rank

```
>>> df.sort_index()
>>> df.sort_values(by='Country')
>>> df.rank()
```

Sort by labels along an axis
Sort by the values along an axis
Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

```
>>> df.shape
>>> df.index
>>> df.columns
>>> df.info()
>>> df.count()
```

(rows, columns)
Describe index
Describe DataFrame columns
Info on DataFrame
Number of non-NA values

Summary

```
>>> df.sum()
>>> df.cumsum()
>>> df.min() / df.max()
>>> df.idxmin() / df.idxmax()
>>> df.describe()
>>> df.mean()
>>> df.median()
```

Sum of values
Cumulative sum of values
Minimum/maximum values
Minimum/maximum index value
Summary statistics
Mean of values
Median of values

Applying Functions

```
>>> f = lambda x: x*2
>>> df.apply(f)
>>> df.applymap(f)
```

Apply function
Apply function element-wise

Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
a    10.0
b     NaN
c     5.0
d     7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
a    10.0
b     -5.0
c     5.0
d     7.0
>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```



Python for Data Science: Linear Regression

```
print(__doc__)

# License: BSD 3 clause

import matplotlib.pyplot as plt

import numpy as np

from sklearn import datasets, linear_model

from sklearn.metrics import mean_squared_error, r2_score

# Load the diabetes dataset

diabetes_X, diabetes_y =
datasets.load_diabetes(return_X_y=True)

# Use only one feature

diabetes_X = diabetes_X[:, np.newaxis, 2]

# Split the data into training/testing sets

diabetes_X_train = diabetes_X[:-20]

diabetes_X_test = diabetes_X[-20:]

# Split the targets into training/testing sets

diabetes_y_train = diabetes_y[:-20]

diabetes_y_test = diabetes_y[-20:]

# Create linear regression object

regr = linear_model.LinearRegression()

# Train the model using the training sets

regr.fit(diabetes_X_train, diabetes_y_train)

# Make predictions using the testing set

diabetes_y_pred = regr.predict(diabetes_X_test)

# The coefficients

print('Coefficients: \n', regr.coef_)

# The mean squared error

print('Mean squared error: %.2f'

      % mean_squared_error(diabetes_y_test,
                           diabetes_y_pred))

# The coefficient of determination: 1 is perfect prediction

print('Coefficient of determination: %.2f'

      % r2_score(diabetes_y_test, diabetes_y_pred))

# Plot outputs

plt.scatter(diabetes_X_test, diabetes_y_test, color='black')

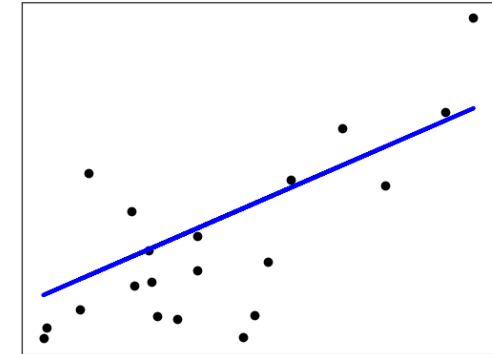
plt.plot(diabetes_X_test, diabetes_y_pred, color='blue',
         linewidth=3)

plt.xticks(())

plt.yticks(())

plt.show()
```

The coefficients, residual sum of squares and the coefficient of determination are also calculated.



Out: Coefficients:
[938.23786125]
Mean squared error: 2548.07
Coefficient of determination: 0.47