T1_1_Introduction_to_Python

January 25, 2021

STAT4609 Example Class 1 (I)

Introduction to Data Science in Python

This notebook helps introduce some of the most basic tools that are commonly used for doing data science and statistics in Python.

Table of contents

Note: you will need to run the following code cell every time you restart this notebook

If this is your first time using Jupyter, click the block of code below and either press the Run button or press `Shift + Enter` on your keyboard.

[3]:
```
from platform import python_version

print(python_version())
```

```
3.7.4
```

[31]:
```
import pandas               as pd
import matplotlib.pyplot    as plt
import numpy                as np
import seaborn              as sns
from   sklearn.linear_model import LinearRegression

from IPython.display import display

iris = sns.load_dataset('iris')
```

1 Jupyter Notebook

Jupyter Notebook is an interactive tool for running code and visualizing data. Each notebook consists of a series of *code cells* and *Markdown cells*.

- Code cells allow you to run code in a number of languages. Behind the scenes, Jupyter runs a "kernel" that processes the code whenever you execute a cell. Since this is a Python notebook, Jupyter is running the IPython kernel. However, kernels also exist for Julia, R, and many other languages.
- Markdown cells display text using the Markdown language. In addition to displaying text, you can write equations in these cells using $\LaTeX$.

To run code, click a code cell (like the one below) and do one of the following: * Press `Shift + Enter` on your keyboard * On the toolbar at the top of this notebook, press the Run button.

[32]:
```
print("Hello, world!")
```

```
Hello, world!
```

You can render a markdown cell in the same way. Double click the text below, and try putting in some of the following items:

```
# This is a large heading!
## This is a smaller heading!
### This is an even smaller heading!
Here is some code: `x = y + z`
And here is an equation: $x = y + z$
```

1.1 Cell magic

The IPython kernel provides some useful tools for programmers, including

- Magic commands, which allow you to do things like look up documentation and past commands that you've run, and
- Building graphical user interfaces (GUIs) to make it easier to interact with your code.

Here's an example of a useful magic command. `?` will look up the documentation for a library, class, or function to help you figure out how to use it. For instance, if I want to learn about pandas DataFrames, I can run the following:

[33]:
```
?pd.DataFrame
```

If you want to see all the magic functions that IPython makes available to you, `%quickref` can give you a high-level overview.

[34]:
```
%quickref
```

1.2 Widgets

IPython and Jupyter Notebook also makes it easy to build widgets, which give you a richer interface with which to interact with the notebook. Try running the code cell below. This code creates two plots, and displays them in adjacent tabs.

[35]:

```
%matplotlib inline

import matplotlib.pyplot as plt
import ipywidgets as widgets
from scipy.stats import norm, linregress

out = [widgets.Output(), widgets.Output()]
tabs = widgets.Tab(children=[out[0], out[1]])
tabs.set_title(0, 'Linear regression')
tabs.set_title(1, 'Normal distribution')

with out[0]:
    # Fit line to some random data
    x = np.random.uniform(size=30)
    y = x + np.random.normal(scale=0.1, size=30)
    slope, intercept, _, _, _ = linregress(x,y)
    u = np.linspace(0, 1)

    # Plot
    fig1, axes1 = plt.subplots()
    axes1.scatter(x, y)
    axes1.plot(u, slope * u + intercept, 'k')
    plt.show(fig1)

with out[1]:
    # Plot the probability distribution function (pdf) of the
    # standard normal distribution.
    x = np.linspace(-3.5, 3.5, num=100)
    p = norm.pdf(x)

    # Plot
    fig2, axes2 = plt.subplots()
    axes2.plot(x, p)
    plt.show(fig2)

display(tabs)
```

```
Tab(children=(Output(), Output()), _titles={'0': 'Linear regression', '1': 'Normal distributio
```

You can create much richer and more complex interfaces that include buttons, sliders, progress bars, and more with Jupyter's ipywidgets library (docs).

2 Pandas

pandas is a Python library that provides useful data structures and tools for analyzing data.

The fundamental type of the pandas library is the `DataFrame`. In the following code, we load the iris flower dataset using the seaborn library. By default, this dataset is stored in a pandas `DataFrame`.

[36]:
```python
iris = sns.load_dataset('iris')

# `iris` is stored as a pandas DataFrame
print('Type of "iris":', type(iris))

# Show the first few entries in this DataFrame
iris.head()
```

```
Type of "iris": <class 'pandas.core.frame.DataFrame'>
```

[36]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Let's get some information about the iris dataset. Let's try to do the following:

1. Find out how many columns there are in the `DataFrame` object, and what kinds of data are in each column
2. Calculate the average petal length
3. Determine what species of flowers are in the dataset
4. Get an overall summary of the dataset

[37]:
```python
# 1. Column labels, and types of data in each column
iris.dtypes
```

[37]:
```
sepal_length    float64
sepal_width     float64
petal_length    float64
petal_width     float64
species          object
dtype: object
```

[38]:
```python
# 2. Calculate the average petal length
iris['petal_length'].mean()
```

[38]:
```
3.7580000000000027
```

[39]:
```python
# 3. Determine which iris species are in the dataset
iris['species'].unique()
```

[39]:
```
array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

[40]:
```python
# 4. Summary of the data
iris.describe()
```

[40]:

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Sometimes we need to extract certain rows or columns of a DataFrame. For instance, in the following code we store each species of flower in its own variable:

[41]:

```
"""
IPython.display is a convenience function that works in Jupyter Notebook
(or, more generally, any IPython-based application) that will show
objects in a nicer way than using print(). We'll use it in this notebook
to show some pandas DataFrames.
"""
from IPython.display import display

"""
Create a DataFrame for each species of flower. I've provided two
methods for creating these DataFrames below; pick whichever you
prefer as they are equivalent.
"""
# Method 1: "query" function
setosa     = iris.query('species == "setosa"')
versicolor = iris.query('species == "versicolor"')

# Method 2: index into the DataFrame
virginica = iris[iris['species'] == 'virginica']

"""
Show the first few entries of the DataFrame corresponding to each species
"""
print('Setosa data:')
display(setosa.head())

print('Versicolor data:')
display(versicolor.head())

print('Virginica data:')
display(virginica.head())
```

Setosa data:

```
   sepal_length  sepal_width  petal_length  petal_width species
```

```
0            5.1          3.5           1.4          0.2  setosa
1            4.9          3.0           1.4          0.2  setosa
2            4.7          3.2           1.3          0.2  setosa
3            4.6          3.1           1.5          0.2  setosa
4            5.0          3.6           1.4          0.2  setosa
```

```
Versicolor data:

    sepal_length  sepal_width  petal_length  petal_width     species
50           7.0          3.2           4.7          1.4  versicolor
51           6.4          3.2           4.5          1.5  versicolor
52           6.9          3.1           4.9          1.5  versicolor
53           5.5          2.3           4.0          1.3  versicolor
54           6.5          2.8           4.6          1.5  versicolor
```

```
Virginica data:

     sepal_length  sepal_width  petal_length  petal_width    species
100           6.3          3.3           6.0          2.5  virginica
101           5.8          2.7           5.1          1.9  virginica
102           7.1          3.0           5.9          2.1  virginica
103           6.3          2.9           5.6          1.8  virginica
104           6.5          3.0           5.8          2.2  virginica
```

To extract a column, we can either use `iris[column_name]` or `iris.iloc[:,column_index]`.

[42]:
```python
"""
Get the first column.

Note: whenever we extract a single column of a pandas DataFrame,
we get back a pandas Series object. To turn it back into a DataFrame,
we add the line `first_column = pd.DataFrame(first_column)`.
"""
first_column = iris.iloc[:,0]
first_column = pd.DataFrame(first_column)

print('First column:')
display(first_column.head())

"""
Get the first through third columns
"""
first_through_third_columns = iris.iloc[:,0:3]

print('First through third columns:')
display(first_through_third_columns.head())
```

```python
"""
Get the 'species' column.
"""
species = iris['species']
species = pd.DataFrame(species)

print('Species column:')
display(species.head())

"""
Get all columns *except* the species column
"""
all_but_species = iris.iloc[:, iris.columns != 'species']

print("All columns *except* species:")
display(all_but_species.head())
```

```
First column:

   sepal_length
0           5.1
1           4.9
2           4.7
3           4.6
4           5.0


First through third columns:

   sepal_length  sepal_width  petal_length
0           5.1          3.5           1.4
1           4.9          3.0           1.4
2           4.7          3.2           1.3
3           4.6          3.1           1.5
4           5.0          3.6           1.4


Species column:

  species
0  setosa
1  setosa
2  setosa
3  setosa
4  setosa


All columns *except* species:

   sepal_length  sepal_width  petal_length  petal_width
0           5.1          3.5           1.4          0.2
```

1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

If you want to create your own pandas `DataFrame`, you have to specify the names of the columns and the items in the rows of the `DataFrame`.

[43]:
```
column_labels = ['A', 'B']

column_entries = [
    [1, 2],
    [4, 5],
    [7, 8]
]

pd.DataFrame(column_entries, columns=column_labels)
```

[43]:

	A	B
0	1	2
1	4	5
2	7	8

3 Introduction to NumPy

NumPy is another Python package providing useful data structures and mathematical functions. NumPy's fundamental data type is the array, `numpy.ndarray`, which is like a stripped-down version of a pandas `DataFrame`. However, the `numpy.ndarray` supports much faster operations, which makes it a lot more practical for scientific computing than, say, Python's list objects.

[44]:
```
import numpy as np

# 1. Create an array with the numbers [1, 2, 3]
x = np.array([1, 2, 3])

# 2. Create a 2 x 2 matrix with [1, 2] in the first row and [3, 4]
#    in the second row.
x = np.array( [[1,2], [3,4]] )

# 3. Create an array with the numbers 0, 1, ... , 9. Equivalent to
#    calling np.array(range(10))
x = np.arange(10)

# 4. Create a 2 x 2 matrix with zeros in all entries
x = np.zeros( (2,2) )

# 5. Get the total number of items in the matrix, and the shape of
```

```
#    the matrix.
num_items    = x.size
matrix_shape = x.shape
```

Besides just providing data structures, though, NumPy provides many mathematical utilities as well.

[45]:
```
### Constants: pi
print('       = %f' % np.pi)
print()

### Simple functions: sine, cosine, e^x, log, ...
print('sin(0) = %f' % np.sin(0))
print('cos(0) = %f' % np.cos(0))
print('e^1    = %f' % np.exp(1))
print('ln(1)  = %f' % np.log(1))
print()

### Minimums, maximums, sums...
x = np.array([1,2,3])
print('Min of [1,2,3] = %d' % x.min())
print('Max of [1,2,3] = %d' % x.max())
print('Sum of [1,2,3] = %d' % x.sum())
print()

### Random numbers: uniform distribution, normal distribution, ...
print('Random numbers:')
print('Uniform([0,1]): %f' % np.random.uniform(0,1))
print('Normal(0,1):    %f' % np.random.normal(loc=0, scale=1))
print('Poisson(1):     %f' % np.random.poisson(1))
```

```
       = 3.141593

sin(0) = 0.000000
cos(0) = 1.000000
e^1    = 2.718282
ln(1)  = 0.000000

Min of [1,2,3] = 1
Max of [1,2,3] = 3
Sum of [1,2,3] = 6

Random numbers:
Uniform([0,1]): 0.193352
Normal(0,1):    -1.833997
Poisson(1):     0.000000
```

NumPy is primarily used to do large-scale operations on arrays of numbers. Because it has C code running behind the scenes, it can do these computations extremely quickly – much faster than you could do with regular Python code. Among other things, with NumPy you can

- add a number to every element of an array;
- multiply every element of an array by a number;
- add or multiply two arrays together; or
- calculate a matrix-vector or matrix-matrix product between arrays.

[46]:
```python
x = np.array([1,2,3])
y = np.array([4,5,6])

print('1 + [1,2,3] =', 1 + x)
print('3 * [1,2,3] =', 3 * x)
print('[1,2,3] * [4,5,6] =', x * y)
print('[1,2,3] + [4,5,6] =', x + y)
print('Dot product of [1,2,3] and [4,5,6] =', x.dot(y))
```

```
1 + [1,2,3] = [2 3 4]
3 * [1,2,3] = [3 6 9]
[1,2,3] * [4,5,6] = [ 4 10 18]
[1,2,3] + [4,5,6] = [5 7 9]
Dot product of [1,2,3] and [4,5,6] = 32
```

4 Additional References

- O Reilly provides a couple of good books that go in-depth about these tools and more:
  - Python Data Science Handbook
  - Python for Data Analysis – this book was published in 2012 and may be slightly dated. However, the author provides some Jupyter Notebooks for free in this repository that you may find helpful.
- Check out the full documentation for Jupyter on the Project Jupyter site.
- Plotting tools:
  - **Matplotlib** (Highly Recommended!)
    - **Documentation**
    - **Tutorials**
  - Plotly
    - Documentation
    - Examples
  - Seaborn [The differences between versions are too large, and thus it is confusing sometimes.]
    - Documentation
    - Introduction
- scikit-learn documentation