

Python for Data Science

Python Version 3.X Is a Leading Platform

- Most popular data science programming language
- NumPy
 - Arrays
- SciPy
 - Math and science functionality
- Pandas
 - Data munging and manipulation
- Neural networks
 - PyTorch
 - TensorFlow

Python Is More Than Data Science

- Supports many OS systems
 - **Never** uninstall the system Python!
 - Only Windows doesn't have a system Python
- General purpose language
- Top five general programming language
- Allows for virtual environments
 - Multiple versions
 - Isolation

Python Resources

- [Main Homepage](#)
 - Version 3.8.5 is current Q4 2020
 - Python Version 2.X is end of life, **do not use**
- [Pandas \(1.1.2\)](#)
- [NumPy \(1.19.0\)](#)
- [SciPy \(1.5.2\)](#)

DataScience@SMU

Python Review

Counting and Indexing

- The biggest “gotcha” for Python is “counting” and intervals
- Python (and most computer science) starts counting from 0 (zero)
 - The first item in a list is “item 0,” first in a row is “row 0,” etc.
 - An example with a Python list:

```
test = ["a", "b", "c"]
```

```
test[0] = 'a'
```

```
test[1] = 'b'
```

```
test[2] = 'c'
```

```
len(test) = 3 #length of list
```

Intervals

- Intervals are **closed** on the **left**
- Intervals are **open** on the right

Example

```
range(0,5)  # a list from 0 to 5
```

returns

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

```
range(5,7)
```

returns

```
[5,6]
```

From previous slide

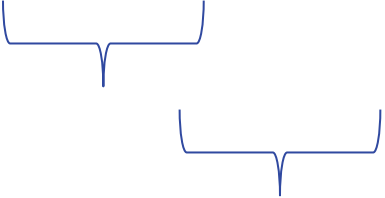
```
test[0:2]
```

returns

```
['a','b']
```


Python Depends on Spaces/Indents

```
for i in range(5):  
    print(i)  
    print(i*i)  
    if i == 4:  
        print("Hi")
```



These spaces **must** be the same, but can be 2, 4, or 5 (typical) or more.

List Comprehensions

Compact way to write a **for** loop and return a list (or dictionary)

```
stuff = []  
for i in list_item:  
    if i > 5:  
        stuff.append(i+1)
```

```
stuff = [i+1 for i in list_item if i > 5 ]
```

(dictionary just uses the {} brackets)

Virtual Environments

- Use Anaconda or venv (virtual environment)
- Allows you to have separate environments with different packages
 - Example
 - Class X requires TensorFlow 1.X
 - Class Y requires TensorFlow 2.1
 - Solution A virtual environment for each allows you to quickly switch between packages without reinstalling
- `conda activate <environment_name>`
- `source <name>/bin/activate`
- `<name>\scripts\activate.bat` (Windows only)

Installation of Packages

- Use the pip package manager

- Works in regular Python and Anaconda

```
python -m pip install <package> #install
```

```
python -m pip --upgrade install <package> #upgrade
```

```
python -m pip install <package>==1.1.0 #version specific
```

Using “python -m” insures that the environment Python is invoked so as to avoid accidentally updating the system Python. Many people omit “python -m,” but using it is a best practice.

```
conda install <package> #anaconda command
```

If a package cannot be found in Anaconda, using pip works.

Also on many systems with Python3, the command is `pip3`.

DataScience@SMU

Pandas Overview

Pandas

- Pandas is the data science workhorse of Python
- Built on top of NumPy
- Facilitates storing and manipulating data in a:
 - Series (1—dimension)
 - Dataframe (2—dimensions)
 - More than 2 dimensions, use a multi-index (rare)

DataFrame Object

- Important: The index is often an integer that corresponds to the row, but it is a **Label**. The index can be anything, including text.
- A DataFrame can have different data types in each column. Common data types are integers, floats, objects (text), and datetime. Even if something looks like a date, **verify**. There is no way to tell by simply looking at the data.

		Column labels			
		timestamp	full_sq	life_sq	floor
Index (row labels)	0	2011-08-20	43	27.0	4.0
	1	2011-08-23	34	19.0	3.0
	2	2011-08-27	43	29.0	2.0
	3	2011-09-01	89	50.0	9.0
	4	2011-09-05	77	77.0	4.0
	5	2011-09-06	67	46.0	14.0

Selecting Data

There are two ways to select data

1. By Label (.loc)
2. By location (.iloc)

```
data.loc[0:3, ["timestamp", "full_sq"]]
```

	timestamp	full_sq
0	2011-08-20	43
1	2011-08-23	34
2	2011-08-27	43
3	2011-09-01	89

Notice 1 less row
as we are not
passing a list of
labels, but a list
of locations!

```
data.iloc[0:3, 0:2]
```

	timestamp	full_sq
0	2011-08-20	43
1	2011-08-23	34
2	2011-08-27	43

Selecting Data (cont.)

You can use conditions to select portions of the DataFrame.

```
data[data['full_sq']>43]
```

	timestamp	full_sq	life_sq	floor
3	2011-09-01	89	50.0	9.0
4	2011-09-05	77	77.0	4.0
5	2011-09-06	67	46.0	14.0

```
: data.loc[data['full_sq']>43, ['timestamp', 'floor']]
```

:

	timestamp	floor
3	2011-09-01	9.0
4	2011-09-05	4.0
5	2011-09-06	14.0

Notice when I sub-select by column, I need to use the .loc notation.

Column Math

- It is possible to do all basic math operations by column.

```
data['full_sq'] + data['life_sq']
```

- You can create columns on the fly.

```
data['new'] = data['full_sq']/data['life_sq']
```

Math on Columns

There are built-in *methods* that can be used on columns.

- Yes, as in class methods. Each **column** is a Series. In fact, it is a Series Object
- The important thing to remember is that methods are used by calling `<name>.method()`

Where `<name>` is the name of your object. Confused? Let's get the sum of a column:

```
data[ 'full_sq' ].sum()
```

Here `data['full_sq']` is a Series Object, and the `sum()` method is called.

Useful Column Summary Methods

- `sum()`
- `mean()`
- `std()`
- `median()`
- `mode()`
- These all return a single value

Other Useful Methods

- `isnull()`
 - Returns a True/False series whether or not the value is null
- `str.contains('<match>')`
 - Returns a True/False series if the entry contains the '<match>'
- `unique()`
 - Returns an array of unique values
- `value_counts()`
 - Returns a DataFrame of values and a count of their occurrence

A Few More for Dataframes

- shape
 - Tells you the dimensions (row,column)
 - **Watch out! No parenthesis!**
- describe()
 - Summary statistics of every columns
- columns, index
 - Lists the column,row names (helpful if you want to rename!)
- dtypes
 - Tells you the data type of each column

DataScience@SMU

Linear Regression

Linear Regression Overview

- This is the base model; many other models build off this concept
- It assumes the output or target is a linear function of your inputs

	timestamp	full_sq	life_sq	floor	price_doc
0	2011-08-20	43	27.0	4.0	5850000
1	2011-08-23	34	19.0	3.0	6000000
2	2011-08-27	43	29.0	2.0	5700000
3	2011-09-01	89	50.0	9.0	13100000
4	2011-09-05	77	77.0	4.0	16331452

$$price_doc = m_1 timestamp + m_2 full_sq + m_3 life_sq + m_4 floor + m_0$$

Solve Your Equation by Minimizing Loss

- We need a loss function.
 - Default is mean squared error.
- Most packages will solve for given loss functions already.
 - Gradient descent is the traditional method.
 - Most packages use faster algorithms.
 - The concept is the same: Find where the loss is minimized by using the first derivative of the loss.

How to Deal with Nonlinear Terms

- If a term is nonlinear, do a variable substitution
- Simple example:

$$y = m_0 + m_1x_1 + m_2x_2^2$$

$$z = x_2^2$$

$$y = m_0 + m_1x_1 + m_2z$$

- In other words, create a new feature/column that is the square of the original and use the new column as the input

Interpretation

- Linear models are very interpretable.
- If your data is normalized, the slopes are the variable importance.
 - Non-normalized data is still valid for the fit; just the interpretation of how important a column is to the output will be skewed by the range on the input
 - Positive slope—direct relationship with output
 - Negative slope—indirect relationship with output
- Many models offer the ability to fit an intercept or not.
- Linear models run **fast**.

sklearn

```
import sklearn
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(x,y)    #x is your input, y is your target
lr.predict(test)    #test is your test input
lr.coef_    # get the slope values
lr.intercept_    # get slope '0'
```

DataScience@SMU