Class 1: Introduction to Python: Devel Environments and Language Basics

Table of Contents / Agenda

1 Python Development Environment

- In Data Science, usually your workflow is interactive
- You need a text editor to write code and a shell to interactively run the code
- You can either have them separately, or use an IDE
- I will only introduce cross-platform tools

1.1 Python Installation

- Python is not domain-specific language
- One needs to install scientific libraries (SciPy Stack)
- Vanilla Python distribution from the official python website does not include essential scientific libraries
- It is highly recommended to use one of the scientific Python distributions such as Anaconda

1.1.1 Anaconda Scientific Python Distribution

- `https://www.anaconda.com/`
- Install many scientific libraries at once
- Manage libraries (install, update) conveniently (Anaconda Navigator)
- Comes with Intel's MKL by default

1.2 Text Editors and IDEs

- When you code, you work with text
- You will spend a lot of time with your text editor

1.2.1 Text Editing Functionalities

- Syntax Highlighting
- Column Editing
- Search/Replace
  - RegEx Search/Replace

1.2.2 Examples

- Notepad, Notepad++
- TextMate

1.2.3 Vim and Emacs

- Vi and Vim
- Emacs
- Editor war
- The Oldest Rivalry in Computing

1.2.4 SublimeText

- Multi selection
  - Suppose I want to change

    ```
    X1, X2, X3, X4 = X1, X2, X3, X4
    ```

  - To:

    ```
    X1, X2, X3, X4 = data.X1, data.X2, data.X3, data.X4
    ```

- Dynamic setting application
- Extensibility

1.2.5 Current Recommendation

- New generation text editors: SublimeText, Atom.io, Visual Studio Code
- Current recommendation: Visual Studio Code
  - Free software and Open Source
  - Powerful
  - Better performance than Atom.io
  - Many extensions (Python tooltip example)

1.2.6 IDE Functionalities

- Code execution
  - Cell support
- Debugging
- Code checking
- Project Management
- Version control integration

1.2.7 IDEs for Python

- Usually runs a Python interpreter within the application
- Tight integration Editor and interpreter
  - Advantage at debugging
- Some candidates
  - Spyder
  - PyCharm
  - WingIDE
  - PyScripter (Windows only)

1.3 Code Snippets Manager

- Code Reuse
  - don’t repeat yourself (DRY) principle
  - c.f. WET solutions: "write everything twice", "we enjoy typing" or "waste everyone’s time"
- You want to accumulate frequently used code snippets for productivity
- Current recommendation: Lepton (Gistbox became not-free)
- Or you can use simpler things such as Simplenote

2 IPython and Jupyter

2.1 IPython and Jupyter

- IPython: Enhanced Python shell. Mainly, it provides
  - tab completion
  - history search
  - on-the-fly documentation
  - `%magic` functions
  - inline plotting
- Documents: `http://ipython.readthedocs.io/en/stable/interactive/tutorial.html`
- Jupyter
  - IPython used to be used to specify both kernel and frontend (IPython, IPython QT Console, IPython Notebook)
  - The frontend part became a language-agonastic separate project
    - e.g., can use Julia and R kernel
  - Now it is Jupyter, which runs an IPython kernel by default
- One kernel, multiple frontend:
  - QT Console
  - Notebook
  - Lab

2.2 Jupyter Lab

- The latest Jupyter frontend: Jupyter Lab
- It is a flexible frontend which can encompass both Notebook and QT Console
- We will use this throughout the semester for class

2.2.1 Running Jupyter Lab

- Run `Jupyter Lab` from `Anaconda Navigator`
  - Optionally you can create a shortcut to `jupyter lab`
- You can run `Anaconda Prompt` and type `jupyter lab <Enter>`

2.2.2 Creating a Notebook

- Create a Notebook
- Notebook consists of multiple cells which can be used for code or other things
- You can insert a new cell with `Insert` menu
- You can run a cell by pressing `Shift+Enter`
- Input the following in the first block:

  ```
  import pandas as pd
  pd.__version__
  ```

- Press `Shift+Enter` to run the code in the cell.

2.2.3 Creating a Console for the Notebook

- Right-click on a cell, select `Create Console for Notebook`.
  - You can have a notebook and a console side-by-side in a browser tab.
- You can rearrange the window layout
- Remember the both notebook and console share the same Python kernel!
- Press `Shift+Enter` to run the code (may change)
  - If you want to change the behavior, see a discussion item on Canvas

2.2.4 Can Open Text and Data (CSV) Files

2.2.5 Workflow - Notebook + Console

- Notebook and Qt Console are standalone programs
- Throughout the semester, we will use Jupyter Lab for clarity

2.3 Convenient Functionalities

2.3.1 tab completion

- The single most convenient functionality
- With a partially completed expression, pressing `TAB` key either completes the expression (when there is an unique expression available) or show candidates

  ```
  >>> pr[TAB]
  ```

2.3.2 history search

- In a console, you can browse the history of commands by `UP` and `DOWN` keys:

```
>>> [UP]
```

2.3.3 On-the-fly documentation

- If you press `Shift+Tab`, it will display documentation about the object under the cursor
- You can put `?` after an object and it will print out documentation

2.3.4 `%magic` functions

- IPython provides many convenient magic functions.
- `%cd`: change working directory
- `%hist`: see history
- `%load`: load a Python script. Test it with an example from `http://matplotlib.org/gallery.html#pie_and_polar_charts`
  - For example,

```
>>> %load http://matplotlib.org/mpl_examples/pie_and_polar_charts/polar_bar_demo.p
```

2.3.5 Inline plotting

- One of the most useful things is that it can show plots inline. Once you run the following magic in Jupyter:

```
>>> %matplotlib inline
```

- Plots will be rendered inline. For example, run a cell with the following:

```
%load http://matplotlib.org/mpl_examples/pie_and_polar_charts/polar_bar_demo.py
```

- This makes the notebook very useful for interactive data exploration.
- You can use `Create New View for Output` as well

2.4 Jupyter QT Console Demo

2.4.1 tab completion

2.4.2 history search

2.4.3 on-the-fly documentation

2.4.4 %magic functions

- You can run a script with %run
- You can load a script from the web with %load:

  ```
  >>> %load http://matplotlib.org/mpl_examples/pie_and_polar_charts/polar_bar_demo.py
  ```

- One of the most useful things is that it can show plots inline. You can run the following magic:

  ```
  >>> %matplotlib inline
  ```

  Then plots will be rendered inline. For example, run the following:

  ```
  >>> %load http://matplotlib.org/mpl_examples/pie_and_polar_charts/polar_bar_demo.py
  ```

2.4.5 inline plotting

3 Python Basics

3.1 Basic Syntax

- = is used for assignment:

  ```
  >>> a = 10  # assign the value 10 to a variable named "a"
  ```

- Python syntax is case-sensitive

  ```
  >>> a  # give me a
  >>> A  # A does not exist
  ```

- Pretty much anything (even unicode in Python 3) can be a variable name

  »> α = 10

  »> α

- No need for a statement terminator (e.g., `;`). `;` is used to supress the value of the last expression. (Mainly for interactive workflow)

  ```
  >>> a
  >>> a;
  ```

- # is used for comments:

```
>>> print(10)  # this is a comment and will be ignored
```

- Function calls always need parentheses, even when there is no argument:

```
>>> print("Hello World!")  # calling print function with argument "Hellow World!"
>>> print()  # calling print function without any argument
>>> print  # shows you the information about the function
```

3.2 Basic Data Types

- You can assign some value to a variable with =:

```
number = 1
```

  - type `number` to verify the value
  - You can use `type()` function to inspect an variable's type

- There are several types of data. The most basic ones are integer, float, and string:

```
number_int = 1
number_float = 1.0
string = "My name is Joon"
```

3.2.1 String

- A string is usually a bit of text
- You can use " and ' interchangeably for strings
  - Useful when you actually have quotes in a string. For example, if the string you want to represents is `"This is an example string"`, then you can use single quotes:

```
string = '"This is an example string"'
```

- You can easily concatenate strings with + operator:

```
string = "My name is"
print(string + ' ' + 'Joon Ro')
```

- Python's string provides a very useful string formatting functionality. If interested, see `https://docs.python.org/3.6/library/string.html`

3.2.2 Built-in Constants

- There are more, but the most frequently used are:

False The false value of the bool type. Assignments to False are illegal and raise a SyntaxError.

True The true value of the bool type. Assignments to True are illegal and raise a SyntaxError.

None The sole value of the type NoneType. None is frequently used to represent the absence of a value

3.3 Lists, Tuples, and Dictionaries

- In addition to the basic data types, there are many data types in Python. e.g., lists, dictionaries, arrays, etc

3.3.1 Lists

- Lists are one of the basic data types, and it is specified with `[]`
- It can hold pretty much anything
- For example:

```
>>> list_example = [1, 2, 'Third', 4, 'Fifth']
```

- In general, you can use `len()` function to get the length of a data:

```
>>> len(list_example)
```

- You always use integer index to access specific value(s) of a list
- In Python, index starts with `0`:

```
>>> list_example[0]  # the first element
>>> list_example[5]  # will give you an error since the last element is 4
```

3.3.2 Tuples

- Similar to lists, but tuples are *immutable*:

```
>>> tuple_example = (1, 2, 'Third', 4, 'Fifth')
```

- Accessing values is the same as lists
- However, you cannot change values
- Again, you can use `len()` to get the length of a tuple

3.3.3 Dictionaries

- You use a dictionary when you want to index an element with a meaningful thing instead of an integer:

  ```
  dict_example = {}
  dict_example['name'] = 'Joon Ro'
  ```

- You can create it like this as well:

  ```
  dict_example = {'name': 'Joon Ro',
                  'major': 'Marketing'}
  ```

3.4 Code Blocks in Python

- In many cases, you have to specify multiple lines of code as a *code block*
- Note that in Python, blocks are distinguished by *spaces*
  - It forces you to indent, which improves readability of code a lot
- For example,

  ```
  if condition is True:
      print("I'm inside the if block")
      # do something

  print("I'm outside of if block")
  ```

3.4.1 Importance of indentation

```
/*  Warning:  bogus C code!  */

if (some condition)
        if (another condition)
                do_something(fancy);
else
        this_sucks(badluck);
```

- Either the indentation is wrong, or the program is buggy, because an "else" always applies to the nearest "if", unless you use braces. (Source: http://www.secnetix.de/olli/Python/block_indentation.hawk)

3.4.2 Readability

- Code is read much more often than it is written
- You will NOT understand the code you wrote before!
- Make sure to:

    1. Comment your code appropriately
    2. Use meaningful variable names
    3. Indent nested code blocks properly

3.4.3 Tab VS. Spaces

- Do not mix tab and spaces
- Using 4 spaces for a tab is recommended

3.5 Conditional Statements and Loops

- Conditional statements and loops are what makes the automation possible
- e.g., loop over each observation in the dataset, and do some calculation depending on whether a variable value satisfies a condition

3.5.1 Conditional Expressions

Meaning	Math Symbol	Python Symbols
Less than	<	<
Greater than	>	>
Less than or equal		<=
Greater than or equal		>=
Equals	=	==
Not equal		!=

3.5.2 `if` .. `elif` .. `else`

- `if` and `elif` will evaluate if the following conditional is `True`. If it is, then it will evaluate the code block associated with it. Otherwise, it will move to the next `elif`, or `else`, or out of the `if` statement

```python
if condition is True:
    print("I'm inside the if block")
    # do something

print("I'm outside of if block")
```

```
a = 10
b = 5

if a > b:
    print("a > b")

elif a < b:  # will not be evaluated if the above condition is true
    print("a < b")

else:  # will not be evaluated if any of the the above conditions is true
    print("a == b")
```

- You can just use a number for the condition in the `if` statements
  - `0` is like `False`. Any number other than `0` will be regarded as `True`

    ```
    if True:
        print("I will always run")

    if 0:
        print("I will never run")
    ```

3.5.3 for loop

- `for` loop will loop over an iterable object and apply the operation inside the block to each element of the object:

```
for counter in (an iterable):
    print("I'm inside the for block")
    # do something

print("I'm outside of for block")
```

- An iterable object is usually a list (but anything can be used)

  ```
  for number in [1, 2, 3]:  # number will take value 1, 2, 3
      another_number = number + 3  # going to be 4, 5, 6
      print(another_number)
  ```

- An useful built-in function: `range()`, which gives you a range of numbers

  ```
  list_numbers_from_range = range(10)  # 10 numbers: 0, 1, 2, ... , 9

  for number in list_numbers_from_range:
      print(number)
  ```

- Often we want to count numbers. For example,

```
list_numbers_from_range = range(10)  # 0, 1, 2, ... , 9

i = 0  # initialize the counter
for number in list_numbers_from_range:
    i = i + 1  # equivalently, i += 1

print(i)
```

3.5.4 Control `for` loop with conditional breaking and continuation

- You can `break` a `for` loop with break:

```
for number in range(10):
    if number > 5:
        break

print(number)
```

- You can also skip one run of the loop with `continue`:

```
sum_numbers = 0

for number in range(10):
    if number > 5:
        continue  # will skip all statements below within the block

    sum_numbers += number

print(number)
```

3.5.5 Simple debugging by raising an exception

- Remember that all the variables will retain their values when the loop stops.

- You can do a simple debugging by forcing an exception:

```
for number in range(10):
    if number > 5:
        1 / 0
```