Basic Python

Note that we will focus on particular aspects of Python that would be important for someone who wants to load in some data sets, perform some computations on them, and plot some of the results. Therefore, we will mostly be talking about Python's built-in data structures and libraries from the perspective of processing and manipulating structured and unstructured data.

Why Python?

The practice of data science involves many interrelated but different activities, including

- accessing data,
- manipulating data,
- computing statistical summaries or business metrics,
- plotting/graphing/visualizing data,
- building predictive and explanatory models,
- evaluating those models, and finally,
- integrating models into production systems

One option for the data scientist is to learn several different software packages that each specialize in one of these things, or to use a general-purpose, high-level programming language that provides libraries to do **all** these things.

Python is **an excellent choice** for this. It has a diverse range of open source libraries for just about everything the data scientist will do. Some of its highlights include:

- **Cross-platform** - high performance python interpreters exist for running your code on almost any operating system (Windows, Mac or Linux) or architecture.
- **Free** - Python and most of its libraries are both open source and free.
- **Simple** - It has efficient high-level data structures and a simple but effective approach to object-oriented programming.
- **Elegant syntax -** which, together with its interpreted nature, makes it an ideal language for scripting and rapid application development in many areas on most platforms.

But wait, Python 2.7 or Python 3.4?

As of December 2015 the latest version of Python is 3.4
However we will be using the old, reliable Python 2.7 for two reasons:

- Python 3 is not backward-compatible with Python 2,
- Many important libraries only work well with 2.7.

The data science community is still firmly stuck on 2.7, which means we will be, too. There is a lot of documentation and support forums available. Make sure to get this version.

Writing Pythonic Code

You will often read questions on Stack Overflow like, ‘What is a more Pythonic way of doing X.’
To know what that means, read **The Zen of Python.** Simply run `import this` on any Python interface.
It is a description of its design principles, and code written using these principles is called ‘Pythonic.’
While there are typically multiple ways to crack a given problem, we will generally favor Pythonic solutions over shabby ones.

Python Package Managers

Do read about **pip** and **conda** - both of which will act as your package/library managers.
To install libraries that aren't part of the Anaconda distribution, you will be using commands such as

- `pip install ggplot`
- `conda install ggplot`

Here we go!

Comments

Any text preceded by the hash mark (pound sign) # is ignored by the Python interpreter.

Whitespace Formatting

Many languages like R, C++, Java, and Perl use curly braces to delimit blocks of code. Python uses whitespace indentation to make code more readable and consistent. A colon denotes the start of an indented code block after which all of the code must be indented by the same amount

One major reason that whitespace matters is that it results in most Python code looking cosmetically similar, which means less cognitive dissonance when you read a piece of code that you didn't write yourself.

Objects, Methods, Attributes and Functions

Every number, string, data structure, function, class, module, and so on exists in the Python interpreter is referred to as a **Python object**.

Each object has an associated

- **type** (int, float, list, dict, str and so on …)
- attached functions, known as **methods**,
  - these have access to the object's internal data.
    They can be called using the syntax: *obj.<method>(parameters)*
- **attributes** which can be accessed with the syntax: *obj.attribute*

"Functions are called using parentheses and passing zero or more arguments, optionally assigning the returned value to a variable: *result = f(x, y, z)*

Modules

Certain functions in Python are not loaded by default.
These include both features included as part of the language as well as third-party features that you download explicitly. In order to use these features, you'll need to **import the modules** that contain them.

> In Python a module is simply a .py file containing function and variable definitions.

You can import the module itself as: `import pandas`
But after this you'll have to always access its functions by prefixing them with the module name,
For example : `pandas.Series()`
Alternatively, we can provide an alias: `import pandas as pd`

This will save us some typing as we can then write `pd.Series()` to refer to the same thing.
Another option is to import frequently used functions explicitly and use them without any prefixes.
For example, `from pandas import Series`

Tip: Importing everything from a module is possible, but is considered bad practice as it might interfere with variable names and function definitions in your working environment.
So **avoid** doing things like: `from pandas import *`

Data Types

Python has a small set of built-in types for handling numerical data, strings, boolean (True or False) values, and dates and time. These include

- *None* - The Python Null Value
- *str, unicode* - for strings
- *int* - signed integer whose maximum value is platform dependent.
- *long* - large ints are automatically converted to long
- *float* - 64-bit (double precision) floating point numbers
- *bool* - a True or False value

You could call the function *type* on an object to check if it is an int or float or string etc.
Type Conversion can be achieved by using functions like `int(), float(), str()` on objects of other types.

Arithmetic using Binary Operators

“Most of the binary math operations and comparisons are as you might expect:”
1 + 23
5 - 7

‘This’ + ‘ That’

Operation	Description
a + b	Add a and b
a - b	Subtract b from a
a * b	Multiply a by b
a / b	Divide a by b
a // b	Floor-divide a by b, dropping any fractional remainder
a ** b	Raise a to the b power
a & b	True if both a and b are True. For integers, take the bitwise AND.
a | b	True if either a or b is True. For integers, take the bitwise OR.
a ^ b	For booleans, True if a or b is True, but not both. For integers, take the bitwise EXCLUSIVE-OR.
a == b	True if a equals b
a != b	True if a is not equal to b
a <= b, a < b	True if a is less than (less than or equal) to b
a > b, a >= b	True if a is greater than (greater than or equal) to b
a is b	**True if a and b reference same Python object**
a is not b	True if a and b reference different Python objects

NOTE that Python 2.7 uses **integer division by default**, so that 5 / 2 equals 2. Almost always this is not what we want, so we have two options:

- Start your files with *from __future__ import division*
- Explicitly convert your denominator to a float as *5/float(2)*

However, if for some reason you still want integer division, use the **//** operator.

Strings

“Many people use Python for its powerful and flexible built-in string processing capabilities. You can write string literal using either single quotes or double quotes, but multiline strings are defined with triple quotes.

```
a = 'one way of writing a string'
b = "another way"
c = """This is a
multiline string"""
```

Strings are

- **sequences** of characters, and so can be treated like other Python sequences (for iteration)
- **immutable,** you cannot modify them in place without creating a new string
- can contain escape characters like \n or \t
  - there’s a workaround if you want backslashes in your string: prefix it with **r** (for **raw**)
- **concatenated** by the + operator, try `'This' + ' and ' + 'That'`

Here I will highlight a few cool **string methods** as a teaser to what you can do with Python

```
my_str = 'a, b, c, d, e'
my_str.replace('b', 'B')
my_str.split(',')
'-'.join(my_str.split(', '))
```

Control Flow with `if, elif, else`

“The if statement is one of the most well-known control flow statement types. It checks a condition which, if True, evaluates the code in the block that follows:

```
if x < 0:
    print 'It's negative'
```

An if statement can be optionally followed by one or more elif blocks and a catch-all else block if all of the conditions are False:

```
if x < 0:
    print 'It's negative'
elif x == 0:
    print 'Equal to zero'
elif 0 < x < 5:
    print 'Positive but smaller than 5'
else:
    print 'Positive and larger than or equal to 5'
```

If any of the conditions is True, no further elif or else blocks will be reached.

We can write **compound logic** using boolean operators like **and, or**. Remember that conditions are evaluated left-to-right and will short circuit, ie, if a True is found in an **or** statement, the remaining ones will not be tested.

```
if 5 < 10 or 8 > 9:
      print ‘The second condition was ignored.’
```

You can also write a **ternary if-then-else** on one line, which sometimes helps keep things concise,

```
      parity = "even" if x % 2 == 0 else "odd”
```

for Loops

These are meant for iteration tasks over a collection (a Python data structure like a Tuple or List.)

Syntax:

```
for value in collection:
    # do something with value
```

Example: Here we print out the squares of the first five natural numbers

```
for x in [1, 2, 3, 4, 5]:
      print x ** 2
```

- The **continue** keyword advances the **for** loop to the next iteration - skipping the remainder of the block.

Example: The following loop sums up values, ignoring instances of **None**

```
total = 0
for value in [1, 2, None, 4, None, 5]:
    if value is None:
        continue
    total += value
```

- The **break** keyword is used to altogether exit the for loop.

Example: This code sums elements of the list until a 5 is reached:

```
until_5 = 0
for value in [1, 4, 2, 0, 7, 5, 1, 4]:
    if value == 5:
        break
    until_5 += value
```

`while` Loops

Python has a while loop as well, which works as expected.

```
x = 0
while x < 10:
    print x, "is less than 10"
    x += 1
```

Python Data Structures are simple, but quite powerful. Understanding them well and mastering their use is critical for a programmer to write efficient code for doing data science. Here we will learn about Tuples, Lists and Dictionaries - each of which are characterized by how data is stored in them, and the use-cases they're most suitable for.

TUPLES

A **tuple** is a sequence of Python objects that is

- one-dimensional,
- fixed-length,
- immutable.

Syntax: We can create tuples in two ways

- A comma-separated sequence of values assigned to a variable (optional: placed inside parentheses)
- Calling `tuple()` on any sequence/iterator (eg. a list)

```
tup = 1, 2, 6
nested_tup = (1, 3, 5), (2, 8), [‘a’, ‘b’]

tuple([1, 2, 7])
tuple(‘forests’)
```

Subsetting: Elements can be accessed with square brackets `[]`, with indexes beginning with 0.

```
nested_tup[0]
```

Immutability: Once a tuple is created, it's not possible to modify which object is stored at an index.

```
nested_tup[1] = (1, 3)
TypeError: 'tuple' object does not support item assignment
```

[*Note*] The **objects** stores in the tuple (eg. a list) might be mutable, so they can be altered (but not moved.)

```
nested_tup[2].append('c')
```

Concatenation: The '+' operator joins tuples to form longer tuples.

```
(4, None, 'foo') + (6, 0) + ('bar',)
```

Tuple Unpacking: In an assignment statement, corresponding elements of a tuple will be assigned to respective objects on the RHS (given that the number of objects is the same as length of the tuple.) This makes is very easy to swap variables.

```
a, b, c = (1, 2, 3)
a, b = b, a
```

Tuple Methods: Press <tab> following a dot after the tuple object. Though there aren't too many Tuple methods, *count* is one of the useful ones.

```
nested_tuple.count()
```

LISTS

A Python **list** is simply an ordered collection of values or objects. It is similar to what in other languages might be called an *array*, but with some added functionality. Lists are

- **one-dimensional**
- containers for collections of objects of **any type**
- **variable-length**
- **mutable**, ie, their contents can be modified

Syntax: They can be defined using

- square brackets `[]` or
- using the `list()` type function
- Python functions that produce lists

```
int_list = [1, 2, 3]
mix_list = ["string", 0.1, True]
list_of_lists = [int_list, mix_list, [‘A’, ‘B’]]
x = range(10)
```

Subsetting

Single elements can be accessed using their index or position

```
x[4]        # fetches the 5th element
x[-1]       # fetches the last element
```

Subsets of lists (smaller lists) can be accessed using **integer slicing**

```
x[:4]       # first four elements
x[4:]       # all elements from fourth to the end
x[-3:]      # last three elements
```

List Methods

1. **Adding elements**

    a. `append()`, to add single elements **at the end** of the list.
    For example: `x.append('a')`

    b. `extend()`, to add multiple element **at the end** of an existing list.
    For example: `x.extend([10, 11, 12])`

    [Note] Lists can be combined/concatenated using the + operator.
    For example: `[1, 2, 3] + ['a', 'b', 'c']`

    c. `insert()`, to insert elements at a specific index (this is an expensive operation)
    For example: `x.insert(3, 'a')`

2. **Removing elements**

    a. `pop()`, removes and returns an element at a particular index (default: from the end)
    For example: `x.pop(5)`

    b. `remove()`, takes an element as input and removes its first occurrence
    For example: `x.remove(5)`

3. **Sorting**
    a. `.sort()`

List Functions

```
x = list('just a string')
```

- `len()`, returns the number of elements in the list
  For example: `len(x)`

- `in`, checks whether an element belongs to a list and returns a boolean
  For example: `'t' in x`

- `sorted()`, returns a new list from the elements of given list, sorted in ascending order
  For example: `sorted(x)`

- `reversed()`, iterates over the elements of a sequence in reverse order
  For example: `reversed(x)`

List Unpacking works a lot like tuple unpacking, where you can assign list elements to objects using an assignment statement.

For example: `a, b, c, d = [1, 2, 3, 4]`

Dictionary (or `dict`)

dict is likely the most important built-in Python data structure. It is a flexibly-sized collection of **key-value pairs**, where *key* and *value* are Python objects.

We frequently use dictionaries as a simple way to represent structured data.

```
doc_info = {
    "author" : "dushyantk",
    "title" : "Data Science in Python",
    "chapters" : 10,
    "tags" : ["#data", "#science", "#datascience", "#python", "#analysis"]
}
```

Syntax: dicts are created using curly braces {} and using colons to separate keys and values.

```
empty_dict = {}
a_dict = {‘k1’:12, ‘k2’:36}
b_dict = {'a': 'a string', 'b' : [1, 2, 3, 4]}
```

Subsetting: We can access or insert the value/object associated with a key by using the curly brackets

```
b_dict[a]              # retrieves ‘a string’
b_dict[‘c’] = 3.142    # adds a new key-value pair to the dict
```

[Note] We can check if a dict contains a key using the **in** keyword

```
‘d’ in b_dict
```

`dict` Methods

1. Removing/Adding elements

    a. `.keys(), .values(), .items()` - will return the keys, values, pairs of the dict
    For example:
    ```
    a_dict.keys()
    a_dict.values()
    a_dict.items()
    ```

    b. `.del()` - Will remove the key-value pair associated with passed key
    For example:
    ```
    del b_dict['a']
    ```

    c. `pop()` - works in much the same fashion
    For example:
    ```
    b_dict.pop('c')
    ```

    d. `.update()` - will merge two given dictionaries
    For example:
    ```
    b_dict.update(a_dict)
    ```

2. Finding elements

    a. `.get()` - behaves gracefully for missing keys. It is used to fetch a value from a dict. If the key is found, it returns the associated value. It returns a default value if the key isn't contained in the dict. This ensures that an exception is not raised.
    For example:
    ```
    b_dict.get('d', 'Nothing found')
    ```

Set

A set is an unordered collection of unique elements. They can be thought of being like dicts, but keys only, no values.

Syntax: A set can be created in two ways:

- via the set function,
- using a set literal with curly braces:

```
set([2, 2, 2, 1, 3, 3])      # produces set([1, 2, 3])
{2, 2, 2, 1, 3, 3}           # produces set([1, 2, 3])
```

Sets support mathematical set operations like union, intersection, difference, and symmetric difference.”