Intro To Python

Objectives

Today's Objectives:

- Python Vs. Other Languages
- Overview of the built-in data types and use cases for each
- Explain the difference between mutable and immutable data types
- What are **generators** and how do they work?
- `if __name__=='__main__'` blocks and why we use them
- What is Pythonic code?
- Work Flow
- Just Enough `bash`
- Installing apps with `brew`

Agenda

- Basic data types/structures:
  - `ints`, `floats`, `strs`
  - lists, tuples, dictionaries, sets
- Mutability/Immutability and Python Data Structures
- Hash Maps and Sets/Dictionaries
- Generators
- Writing Pythonic Code
  - Pep8
- Terminal Shortcuts
- Workflow

Why does this matter?

- Python (data structures, mutability, generators, & c.)
  - Python has become the *de facto* language for data science (*R* is a close second), and thus having a solid understanding of the fundamentals will give you a good base to build off of.
- Pep8 (i.e. writing clean code)
  - People like working with people who write clean code. When you write clean code, your work is easier to follow, digest, and reuse, thereby making you and your team more efficient.
- Terminal Shortcuts & Workflow
  - Having an efficient workflow will save you time. When you spend the majority of your day at your keyboard, 10 seconds here and 10 seconds there adds up pretty quickly.

Python

Duck Typed

Unlike in other languages, where you must explicitly declare what datatype a variable will take on, Python will try to interpret what datatype something should be.

Duck Typed

```
In [1]: 3 / 2
Out[1]: 1
In [2]: print type(3), type(2), type(_)
<type 'int'> <type 'int'> <type 'int'>
In [3]: 3. / 2
Out[3]: 1.5
In [4]: print type(3.), type(2), type(_)
<type 'float'> <type 'int'> <type 'float'>
In [5]: float(3) / 2
Out[5]: 1.5
In [6]: print type(float(3)), type(2), type(1)
<type 'float'> <type 'int'> <type 'int'>
```

White Space Determines Scope

In Python, white space determines the scope of a function. When you declare a function with `def my_func(*args, **kwargs):`, everything that follows that is indented 4 spaces is said to be in the scope of that function.

As such, it is important to set your tab length to 4 spaces in whatever text editor you choose to use and that you are using a plain-text editor (i.e. NOT Word. . .)

Basic Data Types

Basic data types in Python consist of:

1. **ints** - Integers (type `int`)
2. **floats** - Floating point numbers (e.g. decimals)(type `float`)
3. **strings** - Collections of characters (type `str`)
   - Constructed with a **matching** set of single or double quotation marks (" or "")
   - Have to use the double quotation marks if using a contraction (e.g. "I'm using the contraction can't"), or escape the single quotation in the contraction
   - You can also use a set of triple quotation marks to represent a string, but typically only used for multi-line strings

Built-in Data Structures

Built-in data structures in Python consist of:

1. **Lists** - ordered, **dynamic** collections that are meant for storing collections of data about disparate objects (e.g. different types) (type `list`)
2. **Tuples** - ordered, **static** collections that are meant for storing unchanging pieces of data (type `tuple`)
3. **Dictionaries** - unordered collections of key-value pairs, where each key has to be unique and immutable (type `dict`)
4. **Sets** - unordered collections of unique keys, where each key is immutable (type `set`)

Lists

- Lists can be constructed in one of two ways:

1. Using **square** brakets:

```
my_lst = [1, 2, 3, 'multiple', 'data', ('types', 1.0)]
```

2. Using the `list` constructor (takes an *iterable*):

```
my_lst = list([1, 2, 'multiple', 'data', ('types', 1.0)])
```

- Commonly used methods available on a list:
  - `append`: Adds an element to the end of the `list`
  - `sort`: Sorts the list, in-place
  - `extend`: Appends the elements from an `iterable` to the end of the `list`

Tuples

- Tuples can be constructed in one of two ways:

1. Using **standard parentheses**:

```
my_tup = (1, 2, 3, 'multiple', 'data', 'types', 1.0)
```

2. Using the `tuple` constructor (takes an *iterable*):

```
my_tup = tuple([1, 2, 3, 'multiple', 'data', 'types'])
```

- Tuples are meant to be lightweight, and as such only have two methods:
  - `count`
  - `index`
- You should be very careful when storing mutable data types inside of a tuple (we'll see an example of this in a second)

Dictionaries

- Dictionaries can be constructed in three ways:

1. Using **curly** brackets:

```
my_dict = {'Indiana': 'Indianapolis', 'Texas': 'Austin',
           'Colorado': 'Denver'}
```

2. Using `dict` constructor:

```
my_dict = dict(Indiana='Indianapolis', Texas='Austin',
          Colorado='Denver')
```

3. Using `dict` constructor around a list of `(key, value)` tuples

```
my_dict = dict([('Indiana', 'Indianapolis'),
          ('Texas', 'Austin'), ('Colorado', 'Denver')])
```

Dictionaries

- Commonly used methods available on a `dict`:
  - `get` - Takes a `key` from which to return the associated `value`, or a default argument if that `key` is not in the dictionary
  - `iteritems` - Returns a `generator` that can be used to iterate over the `key, value` pairs of the dictionary
  - `update` - Update one `dict` with the `key, value` pairs

Sets

- Sets can be constructed in two ways:

1. Using **curly** brackets:

```
my_set = {1, 2, 3, 4}
```

- NOTE: We can't create an empty set using empty **curly** brackets. This will be interpreted as a `dict`. We would instead do `my_set = set()`

2. Using the `set` constructor

```
my_set = set([1, 2, 3, 4])
```

- Useful for casting a different type of iterable to a `set`, retaining only the unique entries

Sets

- A `set` is primarily used to keep track of unique elements and check membership. `set` objects also have all the standard mathematical operations that you would expect a set to have (e.g. intersection, union, etc.)

Mutability/Immutability

- **Mutable** - Refers to a data structure whose state can be altered after it has been created
  - `list, dict, set`
- **Immutability** - Refers to a data structure whose state cannot be modified after it has been created
  - `int, float, string, tuple`

Lists Vs. Tuples

- A `tuple` is effectively an immutable version of `list`. As such, we typically use a `tuple` when we have data that **will not** change (**fixed** size and **fixed** values), and a `list` when we have data that **might** change (in size or value)
- We should also be very careful whenever storing mutable data types inside of an immutable data type (e.g. storing `lists` inside of a `tuple`). The following example will give an example of some of the unexpected results of doing this...

A += Assignment Puzzler

```
In [1]: t = (1, 2, [30, 40])
In [2]: t[2] += [50, 60]
```

What happens next?

A += Assignment Puzzler

```
In [1]: t = (1, 2, [30, 40])
In [2]: t[2] += [50, 60]
```

What happens next?

A ‘t‘ becomes ‘(1, 2, [30, 40, 50, 60])‘

A += Assignment Puzzler

```
In [1]: t = (1, 2, [30, 40])
In [2]: t[2] += [50, 60]
```

What happens next?

A ‘t‘ becomes ‘(1, 2, [30, 40, 50, 60])‘

B ‘TypeError‘ is raised with the message ‘’tuple’ object does not support item assignment‘

A += Assignment Puzzler

```
In [1]: t = (1, 2, [30, 40])
In [2]: t[2] += [50, 60]
```

What happens next?

A ‘t‘ becomes ‘(1, 2, [30, 40, 50, 60])‘

B ‘TypeError‘ is raised with the message ‘’tuple’ object does not support item assignment‘

C Neither

A += Assignment Puzzler

```
In [1]: t = (1, 2, [30, 40])
In [2]: t[2] += [50, 60]
```

What happens next?

A ‘t‘ becomes ‘(1, 2, [30, 40, 50, 60])‘

B ‘TypeError‘ is raised with the message ‘’tuple’ object does not support item assignment‘

C Neither

D Both **A** and **B**

A += Assignment Puzzler

```
In [1]: t = (1, 2, [30, 40])
In [2]: t[2] += [50, 60]
```

What happens next?

D Both **A** and **B**

A += Assignment Puzzler

```
In [1]: t = (1, 2, [30, 40])

In [2]: t[2] += [50, 60]
---------------------------------------------------------------
TypeError                                 Traceback (most rece
<ipython-input-7-9f190ddb433c> in <module>()
----> 1 t[2] += [50, 60]

TypeError: 'tuple' object does not support item assignment
In [3]: t
Out[3]: (1, 2, [30, 40, 50, 60])
```

Hash Maps

- Both dictionaries and sets are built on top of **hash maps**, which are based off of **hash functions**. From a high-level, a hash function takes an object and transforms it into a unique *hash*, such that `hash(obj1) == hash(obj2)` iff `obj1 == obj2`. This allows us to take in an arbitrary key and associate it with a memory location.
- **Hash Maps** allow dictionaries and sets to achieve lightning fast lookups
  - Rather than search over every element in the data structure, dictionaries and sets are able to go right to the *expected* memory location associated with a given key and check if there is the proper `key-value` pair is stored there
  - The downside of using a hash-map is that it is memory expensive due to `key-value` pairs only occupying $\sim 1/3$ of the hash table

Lists Vs. Sets

- Since `sets` can achieve fast lookup, they are incredibly efficient at checking membership (e.g. is the number 5 in our data structure?)
- This is in stark contrast to lists, which potentially have to look at every element to check membership
- **Use sets whenever you will be checking membership**

Dictionaries Vs. Lists/Sets/Tuples

- Dictionaries are a pretty different data structure from the others - they use `key-value` pairs. As such, we use them whenever we need to store data in that way.
- Their use of **hash maps** does have implications for checking membership though!
  - Use `'Indiana'` in `my_dict` and **NOT** `'Indiana'` in `my_dict.keys()`
  - The former will make use of the **hash map**, while the latter will return the keys as a list and then check membership in that

Generators

- **Generators** - Allow us to build up an iterator that evaluates lazily (only loads values into memory when explicitly needed to perform some calculation/operation)
  - `xrange` is the generator equivalent of `range`
  - `izip` is the generator equivalent of `zip`
  - `iteritems` on a dictionary is the generator equivalent of `items`
- General best practices is to use a generator **unless** we explicitly need a full `list` of our items all at once (for example to pass that `list` to a function that would modify it in place)
- If our data can't all fit in memory at the same time, then we are forced to use a generator (common in image processing or large data applications)

`if __name__=='__main__'` blocks

At a high level, `if __name__==__main__:` blocks allow us to separate our code from the functions contained in a file. That way when we import a file, it only imports the functions and doesn't execute any of the other code contained in the file. . . Consider two files with the following code blocks

```
def my_func(number):
    return number + 2

print "I'm going to print when you import this file..."
```

if __name__=='__main__' blocks

```
def my_func(number):
    return number + 2

if __name__=='__main__':
    print "I'm only going to print when you run the file!"
```

Writing Pythonic Code

- Writing **Pythonic** code means we are using the language in such a way that makes our code more readable while (often) at the same time using Python's power to make your solution more optimal
  - General `for` loops to iterate over lists (instead of indexing in)
  - Using `enumerate` if we need to use the index
  - Using `with` statements when working with files
  - Using `izip` to iterate over two lists at the same time
  - Using a `set` to check membership
  - `list` (and other) comprehensions
    - `squares = [x**2 for x in xrange(1000)]`
  - (`if x:`) instead of (`if x == True:`) or (`if x is not None:`)
  - Leveraging `numpy` and `pandas` (when we get there)

Become a Zen Python Master

```
from zen import clean_code
```

The Zen of Python, by Tim Peters

"Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Flat is better than nested. Sparse is better than dense. Readability counts. Special cases aren't special enough to break the rules. . . ."

Pep8

Pep8 - The Style Guide for Python Code

- We mostly just want to know it's a thing. It deals with spacing, variable names, function names, line lengths, & c.
  - Variable and function names should be `snake_case`, not `CamelCase`
  - New levels of indentation should be 4 spaces, and extraneous white space should be avoided
  - Lines should be no longer than 79 characters
  - Docstrings and comments are always welcomed, but don't be verbose and keep them up-to-date (out of date docstrings are worse than none at all!)
- Takeaways? WRITE CLEAN CODE or Pythonistas everywhere will be displeased!

Bash

Unix - Basic Commands

- Survival command for `Unix`, which *everyone* should be using!

Command	Action
`pwd`	Display current directory (print working directory)
`mkdir`	Create a directory (folder)
`rm`	Remove a file
`ls`	Display the contents of a folder
`cd`	Change Directory
`file`	Display file type
`man`	Get help (display the manual for a command)
`touch`	Create a new, empty file
`less`	Page through a file

Terminal Tips/Tricks - Part 1

- **Tab Completion** - We can type the beginning couple of letters of a file or directory and hit tab to complete it (so long as the file/directory name is unique)
  - This is also a thing in your IPython terminal or notebook
  - Can also use tab completion to see what attributes/methods are available on a variable
- **Up/Down Arrows** - To revisit old commands you can typically press the up arrow, where pressing it multiple times allows you to cycle through all your previous commands
- **Left/Right Arrows** - To navigate single characters in a line, you can typically use your left and right arrows. Holding down your meta key (`option` on a mac) will allow you to navigate through whole words at a time

Terminal Tips/Tricks - Part 2

Hold your `control` key (on a mac) which each of the following letters to achieve the action...

Letter	Action
`u`	Erases input from cursor to the beginning of line
`k`	Erases input from cursor to the end of the line
`a`	Jump to beginning of line
`e`	Jump to end of line
`z`	Suspend a program
`c`	Kill a program
`l`	Clear entire screen

Work Flow

Class Workflow

- Class Workflow:
  1. Fork the daily repository from Zipfian on Gihub
  2. Make any changes you'd like, and add/commit/push them
  3. For **assessments**, issue a pull request

Personal Workflow

- Personal workflow
  - We recommend a text editor/terminal combo
    - ★ While integrated development environments (IDE's) such as PyCharm are nice, the niceties built into them can often keep you from thinking through every aspect of a problem. This can slow you down in the long run.
    - ★ In addition, when we get to the *big data* week, you'll be working on AWS machines, where you'll be forced to work in the terminal. Getting as much practice as you can before we get there will be beneficial.

Personal Workflow - Part 2

- Does it matter what text editor I use?
  - This is an opinionated question, but my opinion is no. As long as you get fluent/comfortable in whatever text-editor you're using, that's most important. That being said, it MUST be a plain-text editor (not Word...)
  - I highly recommend *Atom* because it has an insane number of really nice plugins and it has git integration baked in (it was made by GitHub). If you're already comfortable with another editor, feel free to keep using it.
  - As much as somebody might tell you that you should pick up Vim/Emacs (terminal text-editors), that's probably not a good idea while you're in DSI. They all have a fairly steep learning curves, and the cost/benefit just isn't there (at least while you're in DSI), there are more important things to focus on!

iPython Notebooks

- While iPython notebooks have their place (EDA or sharing your results), they are NOT a place you should be developing in as you will quickly realize when you start executing code cells out of order
- If you are using an iPython notebook, you better damn well have a good reason for it or Cary is going to throw a marker at you from San Francisco!

Operating Systems

- We HIGHLY encourage you to work on a Unix based machine and are happy to assist any Windows users in setting up an Ubuntu partition
- Many of the platforms we will be using were made first for Unix and ported to Windows as an afterthought, and so you will potentially run into weird errors
- Thus, the amount of time you spend setting up an Ubuntu environment now will save you countless hours in the future
- If you choose to continue using Windows, we can't guarantee we'll be able to assist with Windows specific errors

Homebrew

What is Homebrew?

- Homebrew, or `brew`, is the missing package manager for Mac OS X
- This allows us to quickly, and easily, install applications which the proper permissions from the command line
- Works similar to how `apt-get` works on a linux machine
- Homebrew installs packages to their own directory and then symlinks their files into `/usr/local`

How to use `brew`

- First step is to always update brew with the most recent packages by running...
  - `brew update`
- We can search for new packages using...
  - `brew search`
- We can install command line applications such as `cowsay`...
  - `brew install cowsay`
- We can install GUI applications such as Atom using the `cask` prefix...
  - `brew cask install atom`
- We can upgrade all our applications by running...
  - `brew upgrade`

git ABCs...

Figure 1:Always Be Committing!