

Principles and Tools for DSI

Objectives

① Development Tools

- ▶ Workflow
- ▶ Terminal
- ▶ IPython
- ▶ Jupyter Notebook

② Python Refresher

③ Style and Structure

- ▶ Clean and efficient code
- ▶ Pythonic code and PEP 8

④ Pair Programming

Why does this matter?

- Development Tools and 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

- ▶ 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 of

- Style and Structure

- ▶ 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

Development Tools

Workflow

Class Workflow

- Class Workflow:
 - 1 Fork the daily repository from gSchool on GitHub
 - 2 Make any changes you'd like, and add/commit/push them

- 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

- 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!

Terminal

Terminal

- Command line (i.e., text-based) way of interacting with system
- Useful for
 - ▶ `git` commands
 - ▶ Installing Python packages
 - ▶ Running Python scripts
 - ▶ Launching IPython/Jupyter Notebook
- `iTerm2` is a nicer version of `Terminal`
- `tmux` is another great alternative

Basic Unix Commands

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

Command	Action
<code>pwd</code>	Display current directory (print working directory)
<code>mkdir</code>	Create a directory (folder)
<code>rm</code>	Remove a file
<code>ls</code>	Display the contents of a folder
<code>cd</code>	Change Directory
<code>file</code>	Display file type
<code>man</code>	Get help (display the manual for a command)
<code>touch</code>	Create a new, empty file
<code>less</code>	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)
- **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) with 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

IPython

- A feature-rich substitute for the Python shell

IPython - View Docstring

View the docstring for any Python object using `?object`, this is a shortcut for `help(obj)`

```
In [1]: ?map
```

```
Init signature: map(self, /, *args, **kwargs)
```

```
Docstring:
```

```
map(func, *iterables) --> map object
```

Make an iterator that computes the function using arguments from each of the iterables. Stops when the shortest iterable is exhausted.

```
Type:          type
```


IPython - Tab Completion

```
In [2]: ma # press tab
        map          %magic          %matplotlib
        max()         %man
        %macro        %%markdown
```

IPython - Access Previous Results

Use `_` for a variable containing the result of the last executed command:

```
In [1]: 9 * 54
```

```
Out[1]: 486
```

```
In [2]: _
```

```
Out[2]: 486
```

```
In [3]: x = _
```

```
In [4]: x
```

```
Out[4]: 486
```

IPython - More Resources

<https://ipython.org/ipython-doc/dev/interactive/tutorial.html>

Jupyter Notebooks

Jupyter Notebooks

- An interactive notebook for Python based on IPython
- **Great for**
 - ▶ Exploratory Data Analysis (EDA)
 - ▶ Demonstration
- **Terrible for**
 - ▶ Robustly engineered code
- Conclusion
 - ▶ Only use notebooks for simple exploratory tasks
 - ▶ Use IDE such as Atom for actual development

Python Refresher

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:

- ❶ **ints** - Integers (type `int`)
- ❷ **floats** - Floating point numbers (e.g., decimals) (type `float`)
- ❸ **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:

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

Lists

- Lists can be constructed in one of two ways:

- ➊ Using **square** brackets:

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

- ➋ Using the `list` constructor (takes an *iterable*):

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

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

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

- Tuples are meant to be lightweight, and as such only have two methods:
 - ▶ `count`
 - ▶ `index`

Dictionaries

- Dictionaries can be constructed in three ways:

- ➊ Using **curly** brackets:

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

- ➋ Using dict constructor:

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

- ➌ Using dict constructor around a list of (key, value) tuples

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

- 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
 - ▶ `items` - Returns a generator that can be used to iterate over the key, value pairs of the dictionary
 - ▶ `update` - Update one dict with 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

- 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...

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 $\text{hash}(\text{obj1}) == \text{hash}(\text{obj2})$ iff $\text{obj1} == \text{obj2}$. 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

Dictionaries operations

- Looping
 - ▶ GOOD: `for key in dictionary`
 - ▶ BAD: `for key in dictionary.keys()`
- Equality
 - ▶ `dict1 == dict2`
- Checking membership
 - ▶ GOOD: `key in dictionary`
 - ▶ BAD: `key in dictionary.keys()`

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)
- 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!")
```


Style and Structure

Writing Good Code

```
t=1*10**-10**2
m=1*10**2
def f(f, f1, q, t, m):
    i=0
    while ((f(q) > t)
           and
           (i < m)):
        i,q=i+1,q-f(float(q))/f1(q)
    return q

f2 = lambda x:x**2
f3 = lambda x:2*x
print( f(f2, f3, 10, t, m)
      )
```

Writing Good Code

```
def find_zero(f, f_prime, x, threshold=1E-100,
              max_iterations=1E100):
    """Finds the zero of a function (Newton-Raphson)"""
    iterations = 0
    x = float(x)
    while f(x) > threshold and iterations < max_iterations:
        iterations += 1
        x -= f(x)/f_prime(x)
    return x

if __name__ == '__main__':
    def f(x): return x**2
    def f_prime(x): return 2*x
    initial_guess = 10
    print(find_zero(f, f_prime, initial_guess))
```

Writing Good Code

- Code is read more than it is written; style is substance
- Structure your code into functions, modules, and classes
- Follow the DRY principle:
 - ▶ DRY: “don't repeat yourself”
 - ▶ in contrast to WET: “we enjoy typing”
- Read this, a lot: <http://legacy.python.org/dev/peps/pep-0008>
- Also this: <https://google-styleguide.googlecode.com/svn/trunk/pyguide.html>

Writing Efficient Code

- Code that analyzes a lot of data can run out of memory or take forever to complete
- Optimizing your code can be the difference between code that takes a few minutes to run and code that will effectively never finish running
- “Runtime analysis” is a very popular interview topic

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 `zip` 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 range(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...”

PEP 8 - 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. . .
 - ▶ 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!

Pair Programming

Pair Programming - What is it?

- 1 computer, 2 keyboards
- Driver + Navigator

Pair Programming - Why

- Learn more
- Higher quality output
- Good practice
 - ▶ Collaborating and communicating at length about complex problem
 - ▶ Working with different skill sets and personalities
- Increasingly popular in industry

Pair Programming - Core Principles

- Get to know your partner
 - ▶ “What did you think about the lecture?”
- Take turns
 - ▶ Trade driver and navigator roles every 30 minutes
- Listen
 - ▶ If your partner asks “why are we doing that?” then take the time to answer
 - ▶ Don't interrupt
- Be patient
 - ▶ If your partner types something that looks wrong, try to understand it before correcting it
- Be clear
 - ▶ Explaining technical concepts is hard. Practice
- Be humble
- Disagree productively