Introduction to Python and Just Enough Bash

Sean Sall

May 9th, 2016

Objectives:

Today's objectives:

- Gain familiarity with each of the built-in data types/structures in Python, and use cases for each
- Explain the difference between mutable and immutable types, and which are which in Python
- Understand *what* a generator is, *how* it works, and specific use cases
- Understand what it means to write Pythonic code
- Be aware of ways to get around the terminal more efficiently

Agenda

- Basic data types/structures:
  - ints, floats, strings
  - lists, tuples, dictionaries, sets
- Mutablility/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, etc.)
  - Python has become the *de facto* language for data science (*R* is a close second), and having a solid understanding of the fundamentals will give you a good base to build off of. It'll jump start your career as a Pythonista.
- Pep8 (or more generally, writing clean code)
  - For better or worse (probably for the better), people like working with people who write clean code. It makes your code more readable and reusable, ultimately making you and your team more efficient.
- Terminal Shortcuts and Workflow
  - Knowing and having an efficient workflow will save you time. When you spend *potentially* all day at your keyboard, 10 seconds here and 10 seconds there adds up pretty quickly. People like people who are efficient.

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 (not worth it)
     - ★ can technically 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** brackets:

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

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

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

     - ★ Useful for casting a different type of iterable to a `list`
- 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 elements from an `iterable` to the end of the `list`

Tuples

- Tuples can also 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', 1.0)
     ```

     - ★ Useful for casting a different type of iterable to a `tuple`
- Tuples are meant to be lightweight, and as such only have two methods:
  - `count`
  - `index`

Dictionaries

- Dictionaries (as you might have guessed) can also be constructed in two ways:
  1. Using **curly** brackets:

     ```
     my_dct = {'Indiana': 'Indianapolis', 'Texas': 'Austin'}
     my_dct = {} # Creates empty dictionary.
     ```

  2. Using a `dict` constructor:

     ```
     my_dct = dict(Indiana='Indianapolis', Texas='Austin')
     ```

     - ★ Not really used, but possible
- Commonly used `methods` available on a `dictionary`:
  - `get` - Takes a `key` from which to return an 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
  - `iterkeys` - The `key` only equivalent of `iteritems`
  - `itervalues` - The `values` only equivalent of `iteritems`
  - `update` - Update one `dict` with another

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. Python interprets empty brackets as a `dict`.

  2. Using the `set` constructor:

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

     - ★ Useful for casting a different type of iterable to a `set`, retaining only one copy of each potentially unique object.

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

Mutability/Immutability

- **Mutability** - Refers to a data structure whose state can be changed after it has been created
  - lists, dictionaries, sets
- **Immutability** - Refers to a data structure whose state cannot be modified after it has been created
  - ints, floats, strings, tuples

Lists vs. Tuples

- A `tuple` is effectively an immutable version of a `list`. As such, we typically use a `tuple` when we know that 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).
  - If we are worried about anything security related (i.e users of our programs somehow changing the values of our data structures), we might also consider using a `tuple` in this scenario as well.

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** is able to take in an arbitrary key and associate it with a memory location.
  - For a `dict`, these keys are each `key` in the `key-value` pair
  - For a `set`, these are simply the values in the `set`
- **Hash Maps** allow dictionaries and sets to achieve fast lookup
  - 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 anything stored there
    - ⋆ Lists don't have this property. They potentially have to search over the entire structure to find a value or discover it's not there.

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 (e.g. is 5 at spot 1? Is it at spot 2? Is it at spot 3? Shucks.)
- **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 still have implications for checking membership, though:
  - Use (`'Indiana' in my_dct`) and **not** (`'Indiana' in my_dct.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 (SLOW).

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` (similar are `iterkeys` and `itervalues`)
- 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)

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 still need 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

- `import this` :

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. Although practicality beats purity. Errors should never pass silently. Unless explicitly silenced. In the face of ambiguity, refuse the temptation to guess. There should be one– and preferably only one –obvious way to do it. Although that way may not be obvious at first unless you're Dutch. Now is better than never. Although never is often better than *right* now. If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea. Namespaces are one honking great idea – let's do more of those!

Pep8

Pep8 - the Style Guide for Python Code

- We mostly just want you to know it's a thing. It deals with spacing, variable names, function names, line lengths, etc...
  - 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/code are worse than none at all)
- Takeaways? Write clean code, always!!

Unix - Basic commands

- Survival commands for `Unix`:

Command	Action
pwd	Display current directory (print working directory)
mkdir	Create a directory (folder)
ls	Display the contents of a folder
cd	Change directory
file	Display file type
man	Get help (manual)

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 (typically alt on mac) will allow you to navigate through whole words at a time.

Terminal Tips/Tricks - Part 2

- Shortcuts for efficiency - Control Key commands:

Hold your control key with each of the following letters to achieve the action...

Letter	Action
u	Erases input from location to beginning of line
k	Erases input from location 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

Class Workflow

- Class workflow
  1. Fork the daily repository from Zipfian on Github
  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 environment's (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 *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. Most people would agree that as long as you get fluent/comfortable in whatever text-editor you're using, that's most important. E.g. can you get around quickly/efficiently in that text-editor, and do you know some shortcuts there?
  - We recommend *Atom* in this class because it has an insane number of really nice plugins. But, there are also a number of other editors that have really nice plugins. If you're already comfortable with another editor, please feel free to keep using it.
  - As much as somebody might tell you that you should pick up Vim/Nano/Emacs (terminal text-editors), that's probably not a good idea while you're in DSI. They all have fairly steep learning curves, and the cost/benefit just is not there (at least while you're in DSI). There are more important things to focus on.