Python Built-in Data Structures, Functions, and Files

Part 1

Download Code Examples

- http://github.com/wesm/pydata-book

Watch These Videos

- Python Tutorial: Anaconda - Installation and Using Conda
  - https://www.youtube.com/watch?v=YJC6ldI3hWk
- Jupyter Notebook Tutorial: Introduction, Setup, and Walkthrough
  - https://www.youtube.com/watch?v=HW29067qVWk

Data Structures and Sequences

Part 1

Tuple

- A tuple is a fixed-length, immutable sequence of Python objects.
- The easiest way to create one is with a comma-separated sequence of values:

```
In [1]: tup = 4, 5, 6
        tup

Out[1]: (4, 5, 6)
```

- When you’re defining tuples in more complicated expressions, it’s often necessary to enclose the values in parentheses, as in this example of creating a tuple of tuples:

```
In [2]: nested_tup = (4, 5, 6), (7, 8)
        nested_tup

Out[2]: ((4, 5, 6), (7, 8))
```

- You can convert any sequence or iterator to a tuple by invoking `tuple`:

```
In [3]: tuple([4, 0, 2])
Out[3]: (4, 0, 2)

In [4]: tup = tuple('string')
        tup
Out[4]: ('s', 't', 'r', 'i', 'n', 'g')
```

- Elements can be accessed with square brackets [ ] as with most other sequence types.
- As in C, C++, Java, and many other languages, sequences are 0-indexed in Python:

```
In [4]: tup = tuple('string')
        tup
Out[4]: ('s', 't', 'r', 'i', 'n', 'g')

In [5]: tup[0]
Out[5]: 's'
```

- While the objects stored in a tuple may be mutable themselves, once the tuple is created it’s not possible to modify which object is stored in each slot:

```
In [6]: tup = tuple(['foo', [1, 2], True])
        tup[2] = False
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-11b694945ab9> in <module>
      1 tup = tuple(['foo', [1, 2], True])
----> 2 tup[2] = False

TypeError: 'tuple' object does not support item assignment
```

- If an object inside a tuple is mutable, such as a list, you can modify it in-place:

```
In [9]: tup = tuple(['foo', [1, 2], True])
        tup[1].append(3)
        tup

Out[9]: ('foo', [1, 2, 3], True)
```

- You can concatenate tuples using the + operator to produce longer tuples:

```
In [10]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[10]: (4, None, 'foo', 6, 0, 'bar')
```

- Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple:

```
In [11]: ('foo', 'bar') * 4
Out[11]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

- Note that the objects themselves are not copied, only the references to them.

Unpacking tuples

- If you try to `assign` to a tuple-like expression of variables, Python will attempt to unpack the value on the right-hand side of the equals sign:

```
In [12]: tup = (4, 5, 6)
         a, b, c = tup
         b

Out[12]: 5
```

- Even sequences with nested tuples can be unpacked:

```
In [13]: tup = 4, 5, (6, 7)
         a, b, (c, d) = tup
         d

Out[13]: 7
```

- Using this functionality you can easily swap variable names, a task which in many languages might look like:
  - ```
    tmp = a
    a = b
    b = tmp
    ```
- But, in Python, the swap can be done like this:

```
In [14]: a, b = 1, 2
         print('a = ', a)
         print('b = ', b)
         b, a = a, b
         print('a = ', a)
         print('b = ', b)

         a =  1
         b =  2
         a =  2
         b =  1
```
```

- A common use of variable unpacking is iterating over sequences of tuples or lists:

```
In [15]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
         for a, b, c in seq:
             print('a={0}, b={1}, c={2}'.format(a, b, c))

a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

- The Python language recently acquired some more advanced tuple unpacking to help with situations where you may want to “pluck” a few elements from the beginning of a tuple.

```
In [16]: values = 1, 2, 3, 4, 5
         a, b, *rest = values

In [17]: a, b
Out[17]: (1, 2)

In [18]: rest
Out[18]: [3, 4, 5]
```

- This `rest` bit is sometimes something you want to discard; there is nothing special about the `rest` name.
- As a matter of convention, many Python programmers will use the underscore (_) for unwanted variables:

```
In [19]: a, b, *_ = values

In [20]: _
Out[20]: [3, 4, 5]
```

Tuple methods

- Since the size and contents of a tuple cannot be modified, it is very light on instance methods.
- A particularly useful one (also available on lists) is `count`, which counts the number of occurrences of a value:

```
In [21]: a = (1, 2, 2, 2, 3, 4, 2)
         a.count(2)

Out[21]: 4
```