Basic Python

June 19, 2016

1 Python Language Essentials

1.1 Basics (from Python for Data Analysis)

Always start working in an appropriate working environment.

```
In [ ]: import os
        os.getcwd() # Returns the current working directory
        os.chdir('/path/to/directory') # Change the current working directory to 'path/to/directory'.
```

An important characteristic of the Python language is the consistency of its object model. Every number, string, data structure, function, class, module, and so on exists in the Python interpreter in its own "box" which is referred to as a Python object. Almost every object in Python has attached functions, known as methods, that have access to the object's internal contents.

Be very careful with aliasing; instead, when try to assign a new variable the object inside another one, make a copy.

```
In [2]: a = [1,2,3]
        b = a; b

Out[2]: [1, 2, 3]

In [3]: a.append(4); b

Out[3]: [1, 2, 3, 4]
```

To check the type of an object and the exisiting methods for such type:

```
In [10]: isinstance(a, int)
         getattr(a, 'remove')

Out[10]: <function list.remove>
```

Strings with a % followed by one or more format characters is a target for inserting a value into that string (this is quite similar to the printf function in C). As an example, consider this string:

```
In [12]: template = '%.2f %s are worth $%d' #In this string, %s means to format an argument as a string
         #%.2f a number with 2 decimal places, and %d an integer
         template % (4.5560, 'Argentine Pesos', 1)

Out[12]: '4.56 Argentine Pesos are worth $1'
```

A for loop can be advanced to the next iteration, skipping the remainder of the block, using the continue keyword.

```
In [ ]: for value in sequence:
            if value is None:
                continue
            #todo
```

For very long ranges, it's recommended to use xrange, which takes the same arguments as range but returns an iterator that generates integers one by one rather than generating all of them up-front and storing them in a (potentially very large) list.

1.2 Data Structures

1.2.1 Tuples

A tuple is a one-dimensional, fixed-length, immutable sequence of Python objects. The easiest way to create one is with a comma-separated sequence of values. Elements can be accessed with square brackets [] as with most other sequence types. Like C, C++, Java, and many other languages, sequences are 0-indexed in Python.

While the objects stored in a tuple may be mutable themselves, once created it's not possible to modify which object is stored in each slot. Use + to concatenate tuples.

```
In [13]: d = (1,2,3)
         f = (2,3)
         d+f

Out[13]: (1, 2, 3, 2, 3)
```

1.2.2 Lists

In contrast with tuples, lists are variable-length and their contents can be modified. Elements can be appended to the end of the list with the append method. Elements can be removed by value using remove, which locates the first such value and removes it from the last.

```
In [20]: a = [1,2,3]
         a.append(4)
         a.remove(2); a

Out[20]: [1, 3, 4]
```

When concatenating lists, use the extend method, not the + operator.

```
In [21]: b = [2,3,4]
         a.extend(b); a

Out[21]: [1, 3, 4, 2, 3, 4]
```

You can select sections of list-like types (arrays, tuples, NumPy arrays) by using slice notation, which in its basic form consists of start:stop passed to the indexing operator [] (the slicing will stop just before the stop). Either the start or stop can be omitted in which case they default to the start of the sequence and the end of the sequence, respectively. Negative indices slice the sequence relative to the end.

```
In [22]: print(a[0])
         print(a[0:3])
         a[-1]

1
[1, 3, 4]

Out[22]: 4
```

Built-in sequence functions It's common when iterating over a sequence to want to keep track of the index of the current item. A do-it-yourself approach would look like:

```
In [31]: i= 0
         for value in collection:
             # do something with value i += 1
```

```
1
2
3
```

Since this is so common, Python has a built-in function enumerate which returns a sequence of (i, value) tuples:

```
In [38]: for i, value in enumerate(collection):
             # do something with value
```

```
10
21
32
```

zip "pairs" up the elements of a number of lists, tuples, or other sequences, to create a list of tuples and it's a generator.

```
In [28]: a = ['M', 'Ma']
         b = ['R', 'O']
         list(zip(a, b))
```

```
Out[28]: [('M', 'R'), ('Ma', 'O')]
```

1.2.3 Dict

dict is likely the most important built-in Python data structure. A more common name for it is hash map or associative array. While the values of a dict can be any Python object, the keys have to be immutable objects like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable, too). The technical term here is hashability. It is a flexibly-sized collection of key-value pairs, where key and value are Python objects. One way to create one is by using curly braces {} and using colons to separate keys and values:

```
In [40]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}
         d1['a']
```

```
Out[40]: 'some value'
```

You can check if a dict contains a key using the same syntax as with checking whether a list or tuple contains a value:

```
In [41]: 'b' in d1
```

```
Out[41]: True
```

Values can be deleted either using the del keyword

```
In [43]: d1[5] = 'some value'; print(d1)
         del d1[5]; d1
```

```
{'b': [1, 2, 3, 4], 'a': 'some value', 5: 'some value'}
```

```
Out[43]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

The keys and values method give you lists of the keys and values, respectively. In no particular order but both methods will output the things in the same order.

```
In [49]: print(d1.keys()); d1.values()

dict_keys(['b', 'a'])

Out[49]: dict_values([[1, 2, 3, 4], 'some value'])
```

Dicts can be updated with the update method.

```
In [50]: d1.update({'b' : 'foo', 'c' : 12}); d1

Out[50]: {'a': 'some value', 'b': 'foo', 'c': 12}
```

It's common to occasionally end up with two sequences that you want to pair up ele- ment-wise in a dict. Since a dict is essentially a collection of 2-tuples, it should be no shock that the dict type function accepts a list of 2-tuples:

```
In [51]: mapping = dict(zip(range(5), reversed(range(5)))); mapping

Out[51]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

It's very common to have logic like, that can be summarized as:

```
In [ ]: if key in some_dict:
            value = some_dict[key]
        else:
            value = default_value
        value = some_dict.get(key, default_value)
```

1.2.4 Sets

A set is an unordered collection of unique elements that can support set theory operations (intersection, symmetric difference, etc.)

```
In [55]: set([3, 1, 3, 3])

Out[55]: {1, 3}
```

1.2.5 Lists, Set and Dict Comprenhension

List comprehensions are one of the most-loved Python language features. They allow you to concisely form a new list by filtering the elements of a collection and transforming the elements passing the filter in one conscise expression.

```
In [ ]: [expr for val in collection if condition]
        #equivalent for loop
        result = []
        for val in collection:
            if condition:
                result.append(expr)

In [56]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
         [x.upper() for x in strings if len(x) > 2]

Out[56]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Set and dict comprehensions are a natural extension, producing sets and dicts in a idiomatically similar way instead of lists. A dict comprehension looks like this:

```
In [ ]: dict_comp = {key-expr : value-expr for value in collection if condition}
        set_comp = {expr for value in collection if condition}
```

1.3 Functions

There is no issue with having multiple return statements. If the end of a function is reached without encountering a return statement, None is returned.

An alternate and more descriptive name describing a variable scope in Python is a namespace. Any variables that are assigned within a function by default are assigned to the local name- space. The local namespace is created when the function is called and immediately populated by the function's arguments

You can naturally use functions as arguments to other functions like the built-in map function, which applies a function to a collection of some kind:

```
In [64]: import re
         states = [' Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlOrIda', 'south carolina##', 'West
         def remove_punctuation(value):
             return re.sub('[!#?]', '', value)
         list(map(remove_punctuation, states))
```

```
Out[64]: [' Alabama ',
          'Georgia',
          'Georgia',
          'georgia',
          'FlOrIda',
          'south carolina',
          'West virginia']
```

1.3.1 Anonymous lambda functions

Python has support for so-called anonymous or lambda functions, which are really just simple functions consisting of a single statement, the result of which is the return value. They are especially convenient in data analysis because, as you'll see, there are many cases where data transformation functions will take functions as arguments.

```
In [65]: def apply_to_list(some_list, f):
             return [f(x) for x in some_list]
         ints = [4, 0, 1, 5, 6]
         apply_to_list(ints, lambda x: x * 2)
```

```
Out[65]: [8, 0, 2, 10, 12]
```

1.3.2 Generators

Any iterator is any object that will yield objects to the Python interpreter when used in a context like a for loop. A generator is a simple way to construct a new iterable object. Whereas normal func- tions execute and return a single value, generators return a sequence of values lazily, pausing after each one until the next one is requested.

A simple way to make a generator is by using a generator expression. This is a generator analogue to list, dict and set comprehensions; to create one, enclose what would other- wise be a list comprehension with parenthesis instead of brackets:

```
In [71]: gen = (x ** 2 for x in range(10))
         for x in gen:
             print(x)
```

```
0
1
4
9
```

```
16
25
36
49
64
81
```

In [ ]: