Introduction to Python

Surya Duggirala

December 2016

1 Introduction

Python is a very versatile language that can be used for data processing, machine learning/AI, back-end scripting, some web development (see Django framework) and occasionally your main calculator among other things. So just some background, Python was created in the 1980's by Guido van Rossum as a language meant to be easy to read and simple to learn programming with, although it isn't perfect by any means. Reading through the code, you can get a solid idea of how accessible Python really is. It's very similar to the English language and because of that, very easy to pick up. Python provides some built in data structures in the form of sets, dictionaries, lists, arrays, and tuples. We'll go more in depth into those later in the note. One of the most useful features of Python is the ability to use the interactive interpreter in the terminal. It provides a simple way to write and test code. If you're trying to see what happens when you run a function the interpreter can provide you with a definitive answer nearly all the time. I say nearly because while you will get the answer, for some more complex analyses of functions like environment diagrams, you'll want to seek deeper understanding of how computers read and interpret code rather than just a surface level understanding of Python. So, without further ado, let's get into Python!

2 The Python Interpreter

As I mentioned before, the Python interpreter is **very** useful for testing code. In order to access the interpreter in Unix and Unix-like operating systems (MacOS, Linux, BSD, etc...) all you do is:

```
$ python
-- Now you'll see something like this --
-- For Windows type in the path to the directory
   where Python is stored like
   C:\YourName\Projects\Python> --
Python 3.5.2 (default, Oct 11 2016, 05:05:28)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.38)]
```

```
on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Now you can practice and test out whatever you want! If you want to try and test functions and variables that you've written in a file you would run a similar command in the terminal.

```
$ python3 -i ex.py
```

3 Basic Functions and Variables

If you're familiar with any C based languages you know that variables are statically typed, meaning that there's only one "type" of object that you can assign to a variable. This isn't the case in Python. Python is what we call *dynamically typed* which means the interpreter takes care of all the hard work, all you need to do is assign a variable to something. We'll go more in depth into types in later notes but for now all you need to really understand is that variable assignment in Python looks similar to this:

```
>>> my_first_int = 3
>>> my_first_string = "Hello World!"
>>> my_first_boolean = True
>>> print(my_first_int)
3
>>> print(my_first_string)
Hello World!
>>> print(my_first_boolean)
True
```

As you can see, there's not much to do here. Functions can get more interesting though. The basic syntax in python for a function is as follows:

```
>>> def my_first_function(param1, param2):
...     print(param1)
...     print(param2)
...     return param1
...
>>> my_first_function(1, 2)
1
2
1
```

We'll go more in depth into how functions work in later notes, but its important to understand first what we're doing here. We can't really go into more complicated concepts with Python unless we first understand what the code is

saying. We have the "def" statement which signifies that we're writing a function. Python was written in C (in most distributions) and so a lot of the heavy lifting that you would see in languages like Java is unnecessary for you to do because it's already done by the original developers. Our function's name comes next followed by the parameters it uses in parentheses. These parameters can be anything whatsoever. Anything Python offers as a tool can be used. Integers, strings, boolean values, even functions can be "passed" into the function. The catch is that if you write a function with two types interacting, you must be absolutely sure that there isn't a clash. For example, you shouldn't add a string to a boolean. Think about that logically, what does something like " 'hello' + False" even mean? That will 100% of the time result in a TypeError. If you're new to computer science which is probably very likely if you're taking CS61A right now, it is beyond important that you remember to check and double check your functions for silly mistakes like this. It's the equivalent of doing algebra and adding instead of subtracting. When your code base starts growing and you've written more lines than you're ok with reading through to debug, you'll thank yourself for being aware of what you wrote the first time around.

4 Data Structures

Python, as I mentioned before, has its own weaknesses. I'm personally not a huge fan of using Python for work that needs a ton of data structures. I think Java is much more versatile and user friendly as you'll see if you decide to take CS61B in the future. Python, though not as varied in its built in offerings, is still a great language for manipulating data in groups. Lists, tuples, sets, and dictionaries are the most common data structures used in this class, but you should also be aware for future reference that libraries such as NumPy offer a few more options. So let's dissect these data structures to understand some key properties and uses.

1. **Lists**
   (a) Zero-indexed, meaning the beginning of the list is accessed like this:

```
>>> lst = [1,2,3,4,5]
>>> lst[0]
1
>>>
```

   (b) Mutable (Can be changed)
   (c) Iterable (Can be cycled through)
   (d) Lists also have a really helpful property called splicing. Essentially what you do is **create** (very imporant to remember you're actually CREATING a new list) a new list which begins at some point and ends at some point. The syntax looks like this:

```
>>> lst = [1,2,3,4,5]
>>> lst_splice = lst[1:3]
>>> lst_splice
[2,3]
```

The first number signifies the starting point and the second number signifies the point up to which we create the new list. It's important to remember that the last index of the splice (3 in the case of the example) is non-inclusive. We do not add the last signified index into our new list. There is another variation of this which skips over a specified number of elements in the list.

```
>>> lst = [1,2,3,4,5]
>>> lst_even_skip = lst[1:4:2]
>>> print(lst_even_skip)
[2,4]
>>> lst_odd_skip = lst[0:5:2]
>>> print(lst_odd_skip)
[1,3,5]
```

The notation goes as follows, first number signifies the starting point, second number signifies the endpoint index, and the third number signifies how many elements are skipped in the iteration. Lists also have a cool little function "count(elem)" which returns the number of times "elem" shows up.

```
>>> lst = [1,1,1,1,2,3]
>>> lst.count(1)
4
>>>
```

2. **Tuples**

   (a) Immutable

   (b) Since it's not mutable, there are no methods like append or remove.

   (c) ordered and zero indexed. Access of elements at an index is the same as in lists.

   (d) Tuples also have the same "count" function as lists.

   (e) Faster than lists (although you don't need to worry about this in this class)

   (f) Probably best to use when you have a series of constant values that you'll use a fair amount. There are a few ways to create tuples (and this kind of creation extends to lists as well). You can create them in the following ways. Feel free to experiment with this a bit to really master.

```
>>> lst = [1,2,3,4,5]
>>> tup = tuple(lst)
>>> print(tup)
(1, 2, 3, 4, 5)
>>> test_set = {1, 2, 3, 3 ,4, 4, 5}
>>> set_to_tuple = tuple(test_set)
>>> print(set_to_tuple)
(1, 2, 3, 4, 5)
```

3. Some useful methods:
   (a) index(elem) - returns the index of the elem.
   (b) count(elem) - returns the number of
4. **Sets**
   (a) Mutable
   (b) Unordered, meaning you can't index it.
   (c) Can only have one of each element.
   (d) Some key methods include:
      i. update(set) - update the set by putting a new set into it.
      ii. pop() - removes an element at random from the set.
      iii. clear()
      iv. add(elem) - adds an element into the set.

   Sets can be created in a very similar way to lists and tuples.

```
>>> test_set = {1,1,1,1,2,3,4,5}
>>> print(test_set)
{1,2,3,4,5}
>>> tup = (1,1,2,2,3,3,4,4,5,5)
>>> print(tup)
tup = (1,1,2,2,3,3,4,4,5,5)
>>> set(tup)
{1,2,3,4,5}
>>> lst = [1,2,3,4,5,5,5,6,6,7]
>>> set(lst)
{1,2,3,4,5,6,7}
```

5. **Dictionary**
   (a) Pairs a key with a value
   (b) Mutable
   (c) Keys are not immutable. They can't be removed unless you clear all keys and elements.

(d) Useful methods:

i. clear()

ii. copy() - creates a **new** dictionary identical to the old.

iii. update(dict) - add the key value pair associated with dict into the dictionary.

iv. setdefault(elem) - adds a key with a None value into the dictionary if the value doesn't exist or resets the existing key's value with None.

```
>>> test_dict = {'hello': 'world',
... 'this': 'is', 'a':'dictionary'}
```

I would mess around with the dictionary and the attributes associated with it. The 'dir' method is quite useful for things like this. We'll go into that next.

5 The 'dir' method

Whenever you want to see all attributes and functions that an object has, use the 'dir' function. It works as follows:

```
>>> import PIL #Don't stress about
>>> # import statements just yet or
>>> # this library. I'm just using it
>>> # as an example because it has a good layout
>>> # when you pass it into dir. I thought NumPy
>>> # was kind of overkill.
>>> dir(PIL)
['PILLOW_VERSION', 'VERSION', '__builtins__',
'__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__path__', '__spec__',
'_plugins']
>>> # PIL is an image processing library. As you
>>> # can see, there are a series of attributes and
>>> # subclasses within the module. We'll get
>>> # more into this in later notes when we
>>> # go over classes and objects.
>>>
>>> dir(dict)
['__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__',
'__init__', '__iter__', '__le__',
'__len__', '__lt__', '__ne__', '__new__','__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
```

```
'__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'clear', 'copy', 'fromkeys',
'get', 'items', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values']
>>>
>>> dir(set)
['__and__', '__class__', '__contains__',
'__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__',
'__iand__', '__init__', '__ior__', '__isub__',
'__iter__', '__ixor__', '__le__',
'__len__', '__lt__', '__ne__',
'__new__', '__or__', '__rand__',
'__reduce__', '__reduce_ex__',
'__repr__', '__ror__', '__rsub__',
'__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__xor__',
'add', 'clear', 'copy', 'difference',
'difference_update', 'discard',
'intersection', 'intersection_update',
'isdisjoint', 'issubset',
'issuperset', 'pop', 'remove',
'symmetric_difference', 'symmetric_difference_update',
'union', 'update']
>>>
>>> dir(tuple)
['__add__', '__class__', '__contains__', '__delattr__',
'__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmul__',
'__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'count', 'index']
>>>
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__',
```

```
'__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy',
'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

6 Errors and Error Handling

1. Common Errors

   (a) **SyntaxError**: Called when your code isn't written properly. For example something like this:

```
>>> def func(x):
...     if x == 4
  File "<stdin>", line 2
    if x == 4
           ^
  SyntaxError: invalid syntax
```

   The issue here is that the if statement is missing a colon. The proper way to write this statement would be:

```
>>> def func(x):
...     if x == 4:
...         print("Hello World!")
...
>>> func(4)
Hello World!
```

   These are obnoxious errors because your logic can be right in its entirety but you forgot to add a colon or a parentheses somewhere. The interpreter looks for specific 'tokens' to understand what the code is saying as you'll learn towards the end of the semester. Without specific tokens, your code is useless. Be careful while reading through your code and make sure everything is in order.

   (b) **ZeroDivisionError**: Simple, you divided by zero somewhere in your code. Find that and the error is gone.

   (c) **IndexError**: You're trying to access an element from a list in an index that doesn't exist. For example:

```
>>> lst = [1,2,3,4,5]
>>> print(lst[5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Be careful about how you index lists and tuples.

(d) **AttributeError**: This happens when you try to get an attribute from a class that doesn't have that attribute. We'll go more into attributes later on in the notes.

(e) **NameError**: This is just saying that the variable you tried calling hasn't been defined yet. Python doesn't let you just set variables aside for use later like in java (int num;). Each variable you have must be assigned to a particular value.

(f) **StopIteration**: Used with iterators. We'll get to this later, it's actually quite useful.

(g) **IndentationError**: Your indentation is off. As I've mentioned before, the interpreter looks for specific characteristics in your code. If those characteristics are off, the interpreter won't know how to read what you wrote.

(h) **TypeError**: You'r trying to relate two or more objects that don't work together. For example:

```
>>> "Hello World!" + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Be careful with your variables, if you get this error you're either blatantly doing something illogical or you're accidentally changing a variable's value in your code so that it can't work with whatever you're doing. It's a pretty generic error, it can take a bit of digging to see really what's going on. Don't be overwhelmed!

(i) **AssertionError**: An assert statement is catching something that shouldn't be there.

```
>>> def func(x):
...     assert x == 5
...
>>> func(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in func
AssertionError
```

You can also add messages after the assertion error like this.

```
>>> assert False == True, "False is definitely not true"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: False is definitely not true
```

(j) **KeyError**: Just saying the key doesn't exist in a dictionary.

(k) **RecursionError**: You've exceeded the recursion depth. This means that 99.9999999% of the time your base case is wrong. Your function is running an "infinite" number of times. Check your base case.

(l) **UnboundLocalError**: You're referencing a variable from outside the scope of your function. We'll go more in depth into this stuff when we get to global and local statements in later notes.

A good way to approach error debugging is to check out the message and line that the error is happening in. Test cases will always show you what errors are happening and what lines they're happening in. This is a great place to start. Once you figure out the exact place that your code is bugging out in, you can follow the train of thought that led to the code to see what exactly is happening and what you did. It's important for you to look through everything linearly. Code runs linearly. As convoluted as it can be sometimes, it will always run as a series of instructions one after the other.

This was a pretty high level run through of Python. As the course progresses you'll get a deeper understanding of these topics and how they fit into the greater scope of computer science.

7 sources

@kennethlove. "What Is Python Used For?" Treehouse Blog. N.p., 18 Aug. 2016. Web. 26 Dec. 2016.