Python 101

Field Services Business Intelligence Internship

Christopher J. Adkins

Quick Overview

The Interactive Shell It's always a good thing to ask why. For example, why are we going over python right now, can't we just go out for a drink? Sadly not right now, but maybe in a few hours. One of the main benefits to python is it's an **interpreted language**. This basically means that we don't compile code, we interpret it with an interpreter (a precompiled shell). Call the shell and try executing some of the code snippets below (I've included the output as well):

```
>>> 1 + 1
2
>>> print("hello world")
hello world
>>> x = 5
>>> y = 2
>>> x/y
2.5
>>> x = 5
>>> x//y
2
```

feel free to try out whatever arithmetic operations you'd like and see how they behave.

HelloWorld.py Now know how the code is being read, let's start with our very first script. It's almost tradition to implement a hello world as your first foray into a new language. So let's write it out and run it:

```
# this is a comment, the interpreter will ignore everything after the # on this line
def main(): # <-- this is a function called main that takes no parameters
  msg = "hello world" # <-- this is a variable called msg, that stores the string "hello world"
  print(msg) # this will print the string "hello world"

if __name__ == "__main__": # this is an if statement, the inside block is run if the equation is true.
  main() # this calls the function we defined above.
```

Run the script using, "python HelloWorld.py". This may be slightly more complicated then you were thinking for hello world, but I cannot understand the importance of structuring your code well. When we run the HelloWorld.py script, it becomes the top level script and defaults to the name __main__. Thus the if statement is satisfied, and we run the function main() which gets us the print statement of "hello world".

Modules / PyPI(Python Package Index) A module is a collection of classes, functions and global variables (e.g. HelloWorld.py is a module with one function called main, we'll get to classes and global variables later). A package is a collection of modules that usually are working together to create a layer of abstraction in your code / use functions and methods that simplify your code. E.g. I want to explore a data set that's saved as a csv(comma separated values), but I don't know how to load it... I can quickly check Stack Exchange and find out that "pandas" is a package designed to handle all of that for me. Now that I know, I can download pandas via the default package management system **pip**. Via the command line we can type:

```
pip install pandas
```

I can access this package by importing the code into my current shell via the **import** command:

```
import pandas as pd # <-- industry standard shorthand for pandas
#rest of my code
```

Since you've downloaded anaconda, you already have most the packages you'll need.

IPython / Jupyter Now wouldn't it be nice if we could write a script but have the flexibility of the Interactive shell. It turns out a bunch of lovely people got together and created just that. So let's launch our first notebook to start playing around and have an easy to persist variables to memory. Start up a session via the command line with:

```
jupyter lab
```

Data Types / Variables Python is a **dynamically typed** language. This means that you , the programmer, don't have to declare the type of a variable before (compile/run-time). The shell does that for you. It may also be useful to know that python is **strongly typed** as well (not weak), which means this means you'll have to explicitly cast any type changes. For example of a weakly typed language, think javascript...

```
>>1+'1'-1
10 // <-- the type of the output is a number
```

In this section we'll go over the basics of main data types (including type hints which you may use an optional static type checker if you're looking to conform to a static practice e.g. mypy).

```
>>> integer:int = 1 #immutable
>>> type(integer)
<class 'int'>
>>> number:float = 1.0 #immutable
>>> string:str = "hello" #immutable
>>> array:list = [integer,string] #mutable
>>> array[0] #index accessor
1
>>> dictionary:dict = {'key1':'value1','key2':'value2'} #mutable
>>> dictionary['key1'] #key accessor
```

```
'value1'
>>> boolean:bool = True #immutable
>>> def annotated(x:int,y:str) -> bool:
...   return x < y
...
>>> point:tuple = (0,1) #immutable
```

Python's Operators Here's a summary of the main operators built into python. Don't forget you can always create your own or override the existing ones! Let $a = 10$ and $b = 20$.

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	$a + b = 30$
- Subtraction	Subtracts right hand operand from left hand operand.	$a - b = -10$
* Multiplication	Multiplies values on either side of the operator	$a * b = 200$
/ Division	Divides left hand operand by right hand operand	$b/a = 2$
% Modulus	Divides left hand operand by right hand operand and returns remainder	$b\%a = 0$
** Exponent	Performs exponential (power) calculation on operators	$a**b = 10^{20}$
// Floor Division	Division where the digits after the decimal point are removed.	$9//2 = 4$
==	If the values of two operands are equal, then the condition becomes true.	$(a == b)$ is not true.
!=	If values of two operands are not equal, then condition becomes true.	$(a! = b)$ is true.
<>	If values of two operands are not equal, then condition becomes true.	$(a <> b)$ is true.
>	If the left is greater than the value of right, then condition becomes true.	$(a > b)$ is not true.
<	If the right is greater than the value of left, then condition becomes true.	$(a < b)$ is true.
>=	If the right is greater than or equal to the value of left	$(a >= b)$ is not true.
<=	If the left is greater than or equal to the value of right	$(a <= b)$ is true.
= Assignment	Assigns values from right side operands to left side operand	$c = a + b$
+= Add AND	Add the right to the left operand and assign the result to left operand	c += a
*= Multiply AND	Multiplies the right to the left operand and assign the result to left operand	c *= a
-= Subtract AND	Subtract the right to the left operand and assign the result to left operand	c -= a

The list goes on with bitwise, identity ops, etc.

Conditional Statements / Switches The **if** command is basically a function that will execute a block of code if the input is True. You can stack these checks with the else if (**elif**) command as well, with a **else** block running if none of your conditional statements are satisfied. What will happen with the following code block:

```
x = 1
y = "yes"
z = True

if (x != y):
  print("x doesn't equal y")
```

```
elif(z or False):
  print("z must be true")
else:
  print("None of my statements were satisfied")
```

Be careful with your data types, we see that when equatable operator (==) was called, the types didn't align and no operator existed to check if they were equal or not. Sometimes you'll have very structured code, where you have clearly defined states to move to. Other languages like C have built in switch systems which look something like:

```
switch(n) {
  case 0:
    printf("You typed zero.\n");
    break;
  case 1:
  case 9:
    printf("n is a perfect square\n");
    break;
  case 2:
    printf("n is an even number\n");
  case 3:
  case 5:
  case 7:
    printf("n is a prime number\n");
    break;
  case 4:
    printf("n is a perfect square\n");
  case 6:
  case 8:
    printf("n is an even number\n");
    break;
  default:
    printf("Only single-digit numbers are allowed\n");
  break;
}
```

2 ways of replicating this type of flow would be with dictionaries and conditional statements:

```
if(n == 0):
  print ("You typed zero.\n")
elif(n== 1 or n == 9 or n == 4):
  print("n is a perfect square\n")
elif(n == 2):
  print("n is an even number\n")
elif(n== 3 or n == 5 or n == 7):
  print("n is a prime number\n)"
```

```
options = { 0 : zero,
           1 : sqr,
           4 : sqr,
           9 : sqr,
           2 : even,
           3 : prime,
           5 : prime,
           7 : prime,
}

def zero():
  print("You typed zero.\n")

def sqr():
  print("n is a perfect square\n")

def even():
  print("n is an even number\n")

def prime():
  print("n is a prime number\n")

options[n]()
```

Notice that we can reference functions as variables, this is situationally useful.

Loops In python, your basic loop is a foreach loop. You're able to iterate through a collection's items. E.g.

```
  numbers = [1,10,20,30,40,50]
  total = 0
  for number in numbers:
    total += number
  print(total)
```

Older styles of for loops (do loops) used to rely on counters or indexes. Which can written as:

```
  numbers = [1,10,20,30,40,50]
  total = 0
  for n in range(len(numbers)): # [0,1,2,3,4,5]
   total += numbers[n]
  print(total)
```

This type of loop will iterate through every element of the collection once, then exit once you've reached the end of the collection. Sometimes you want to create an infinite loop or a loop dependent on a certain parameter and continuously cycle while a condition is true. These are called while loops and will exit once a condition is false or you **break** the loop.

```
while True:
  answer = input("Type something... ")
  if(answer in ("q","quit")):
    break
  print("You wrote %s" % answer)
```

Here's an example of a common math function you've probably heard of.

```
def factorial(n:int) -> int:
  if(n < 0):
    raise Exception("n must be non-negative")
  elif(n == 0):
    return 1
  else:
    product = 1
    while n > 0:
      product *= n
      n -= 1
    return product
```

Can you think of another way of shortening the code for this function? Think recursion.

Flags / Command Line (CL) Arguments If you're using various open source tools and modules, you'll undoubtedly come across tools that have CL arguments or flags. Let's quickly go through how this would look in a script:

```
import sys, getopt

def main(argv):
  inputfile = ''
  outputfile = ''
  try:
    opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
  except getopt.GetoptError:
    print('cli_test.py -i <inputfile> -o <outputfile>')
    sys.exit(2)
  for opt, arg in opts:
    if opt == '-h':
      print('cli_test.py -i <inputfile> -o <outputfile>')
      sys.exit()
    elif opt in ("-i", "--ifile"):
      inputfile = arg
    elif opt in ("-o", "--ofile"):
      outputfile = arg
  print('Input file is %s' % inputfile)
  print('Output file is %s' % outputfile)

if __name__ == "__main__":
  main(sys.argv[1:])
```

Note you may also accomplish the same file utilizing the "argparse" module.

Reading and Writing Files There are a few ways to read and write data and files in python, but let's start by creating a simple text file.

```
file = open("textFile.txt","w")

file.write("Hello World\n")
file.write("This is our new text file\n")
file.write("and this is another line.\n")
file.write("Why? Because we can.\n")

file.close()
```

the "w" stands for write. To read this file we've just created, we'll now use read ("r") mode.

```
file = open("textFile.txt","r")
print(file.read())
file.close()
```

you can also pass a int to read() to specify the number of characters you'd like to read. There are various was to work with raw data using these parsing methods, but we'll finish by showing good practices using **with**.

```
with open("textFile.text") as file:
  for line in file:
    print(line)
```

using with we don't need to call our file closure and looks a little cleaner. Other modes which may be useful to you are "a" and "r+" which allows for file appending and read/write concurrently.

Exercises Here are various questions/task that will test your knowledge of the basics of python.

1. Complete a few string related functions. Finish string1.py.(15 mins)
2. Complete a few list related functions. Finish list2.py.(15 mins)
3. Word distribution of a text file. Provided by Google.(20 mins)
4. Write a simple username and password storage program that will allow you to add/read/edit/delete usernames/passwords to a data file (you could use a .txt) via the command line. See below for example usage ideas (30mins)

   ```
   $chris>accountManager --add <service> <username> <password>
   ```

```
account information saved

$chris>accountManager <service>
<username> <password>

$chris>accountManager --edit <service> <username> <password>
account information modified

$chris>accountManager --delete <service>
<service> account information deleted
```

Objects / Classes The best way to understand an object, is to see one. Here's an example.

```
class A:
  __init__(self):
    self.Property1 = "I am important"
```

Global Context An example of a global variable

```
def bob():
  global me
  me = "locally defined" # Defined locally but declared as global
  print me

bob()
print me    # Asking for a global variable
```

Working with Data We'll mostly be using pandas and dataframes to manipulate data. Next time we'll be diving deeper in various packages for graphing / server capabilities.