Python Crash Course

By Corey McMillan

First we need to make sure python3 is installed.

Type: *python3 –version* to see if python3 is installed (it should be)

If python3 is not installed type: *sudo apt-get install python3*

To open the python interpreter type *python3* and hit enter.

You should see some information on the version of python you are using and your cursor will be at the end of a line starting with >>>. This is where we will type our python commands.

Arithmetic

Python can be used to handle simple calculations. Simple operators (+, -, *, /, %) work just as they do in other programming languages, and order of operations (PEMDAS) applies. Try entering a few now. At the end of each line hit enter and you will be presented with the answer, for example:

```
>>> 2 + 2
4
>>> 7 * 2
14
```

Division will always return a float (a decimal number), for example:

```
>>> 7 / 2
3.5
```

Using // will perform floor division, removing the decimal remainder form any division, for example:

```
>>> 7 // 2
3
```

Using % will return the remainder of division, for example:

>>> 7 % 2

1

In Python, you can use ** to calculate powers of numbers, for example:

*>> 7 ** 2*

49

The equal sign = is used to assign a value to a variable, for example:

>>> number = 123

When you assign a value to a variable you will not receive any result on the next line, however entering the variable name in the interpreter will return its value, for example:

>>> number

123

Strings

You can make string in python using double or single quotes:

>>> 'Hello python!'

Hello python!

>>> "And with double quotes!"

And with double quotes!

You can concatenate strings by using the + operator, for example:

>>> 'Hello ' + 'world!'

Hello world!

You can also repeat string any number of times using the * operator, for example:

*>>> 'Baby ' * 3*

Baby Baby Baby

String can be indexed, and accessed using [], for example:

>>> word = 'Python'

>>> word[0]

P

This notation with [] at the end is the notation for a list, a type of data structure that contains a list or collection of items. We'll look at lists a bit later.

You can also access strings starting from the end, in reverse using negative indexes, for example:

>>> word[-1]

N

>>>word[-5]

Y

Along with indexing, string can be *sliced*, allowing you to obtain substrings, for example:

>>> word[0:3]

Pyt

>>> word[3:5]

ho

Notice how the second number is excluded while the first number is included.

You can also omit the first or second number in the slicing operation,

>>> word[3:]

hon

>>> word[:5]

pytho

Data Structures

One useful data structure in python is the list. This is similar to the array in other languages such as java, c++, c#, and many others. You can create a list by typing the following into your console:

>>> num = [1,2,3,4,5,6,7,8,9]

Typing 'num' into your console and pressing enter will produce the contents of the list.

[1, 2, 3, 4, 5, 6, 7, 8, 9]

You can access individual elements of a list by using the [] bracket notation from earlier. So if you wanted to access the first element, you could type:

>>> num[0]

And you would receive a 1 as your output.

You can add numbers to the end of a list by using the .append() function:

>>> num.append(10)

>>> num

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Lists can contain different types of data

>>> mixed = [1, 'string', 5.0]

Control Flow: while, for, if, elif, break, continue

Python, like most other programming languages has control flow statements that allow you to execute different parts of code depending on the situation.

While

Lets take a look at the while statement. First lets define a variable, x, and initialize it to 5. Then type the following:

>>> while x < 10:

… print(x)

… x += 1

Then press enter. You should see 5 through 9, printed in the console. The while statement executes the code inside the block *while* the statement is true. As soon as the statement is evaluated to false, the code skips over it and performs whatever code is written after. In our code we printed the value of x, then incremented x by 1 each time the block was executed. After we incremented x to 10, the code evaluated that x was no longer less than (<) 10, so the block did not execute again. While statements are good when you need something to happen a number of times, and

the statement can be evaluated to false at some point. If the while block is never evaluated to false the code will run forever, creating an infinite loop, which is bad.

For

The for statement executes code for the number of times there are a set number of items, such as working with lists. If we want to add 2 to every number in our list from earlier, for instance, we could do this by typing

>>> for n in num:

… print(n + 2)

…

Range()

You can also iterate over a sequence of numbers using the range() function

>>> range(10)

[0,1,2,3,4,5,6,7,8,9]

Notice how 10 is left out in the output.

You can give 2 arguments to the range function to go from the first number to 1- the second number

>>> range(5,10)

[5,6,7,8,9]

You can even provide 3 arguments to the range function to specify how much to increment by

>>> range(0,10,3)

[0,3,6,9]

Notice how the third argument tells the function to increment by 3 each time instead of just 1.

Break and **Continue**

You can use the **break** keyword to break out of a looping or iterating statement

```
>>> for n in list:
...     if n == 8:
...         break
...     print('This will execute ' , n)
...
This will execute 1
This will execute 2
This will execute 3
This will execute 4
This will execute 5
This will execute 6
This will execute 7
```

Once the statement evaluates to true, the for statement exits, and 9 is never reached

You can use the **continue** statement to continue on to the next iteration of the loop

```
>>> for num in range(0,15):
```

```
...     if num % 2 == 0:
...         print('Even number!', num)
...         continue
...         print('This wont run')
...     print('Found a number!', num)
...
Even number! 0
Found a number! 1
Even number! 2
Found a number! 3
Even number! 4
Found a number! 5
Even number! 6
Found a number! 7
Even number! 8
Found a number! 9
Even number! 10
Found a number! 11
Even number! 12
```

Found a number! 13

Even number! 14

Pass

The pass statement does absolutely nothing. What a bum! It can be useful, however, whenever a statement is required, but the program doesn't require that anything actually be done. This can be useful when you're outlining a function or class but don't want to implement it yet. Speaking of defining a function…

Functions

We've already used functions, such as range() or len(), but what is a function? A function is any piece of code that can be reused to perform a specific purpose or FUNCTION. They're really useful when you need to perform the same operations frequently, and you can define your own functions to cut down on the amount of code you have to write. Take adding two numbers together for example

```
>>> def add_two(a, b):
...     return a + b
...
>>> add_two(10, 20)
30
```

As long as the operation makes sense, you can even use different data types

```
>>> add_two('Hello ', 'world!')
'Hello world!'
```

>>> add_two([1,2,3], [4,5,6])

[1, 2, 3, 4, 5, 6]

Any type of operation that you can perform, you can put into a function.

Classes

Classes are useful when you want to represent an object. Think of a class as a type of blueprint that you can specify attributes and certain behaviors. Try entering this in nano and saving it as Car.py

```
class Car:
    def __init__(self):
        self.make = 'Honda'
        self.model = 'Civic'
        self.doors = 4
        self.spinners = False

    def print_specs(self):
        print(self.make, self.model, self.doors, self.spinners)

    def change_make(self, make):
        self.make = make

    def change_model(self, model):
        self.model = model
```

once you have finished type

from Car import Car

This allows you to use your Car class in the command line. Now make a new Car object

>>> x = Car()

And use your methods by typing

>>> x.print_specs()

Honda Civic 4 False

Try using the methods from the car class to change your car object's specs. You can even make another car. Heck, make a whole garage!

Classes can be inherited from other classes, giving them the attributes and methods from the inherited class. Type *nano SportsCar.py* and copy this into the editor

```
from Car import Car


class SportsCar(Car):
    def __init__(self):
        super().__init__()
        self.make = 'Nissan'
        self.model = 'GTR'
        self.spinners = True

    def go_fast(self):
        print('Skrtskrrrrt')
```

after saving, type

>>> from SportsCar import SportsCar

Now make a new SportsCar object

>>> fast = SportsCar()

Now try using the old methods from the Car class on your fancy new sports car

>>> fast.print_specs()

Nissan GTR 4 True

Notice how even though we didn't explicitly give the SportsCar class a print_specs method it is able to inherit it from the Car class and use it like it was it's own.

Lastly, I'll leave you with the Zen of Python

>>> import this