python_basics

September 3, 2020

1 Intro to Python Basics

1.0.1 We're going to start with some Python basics before we get into the "financial/data science Python basics".

https://learnxinyminutes.com/docs/python/ https://www.w3schools.com/python/default.asp

These links can be helpful resources when beginning to learn Python.

[1]:
```
import sys
print(sys.version)
```

```
3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AMD64)]
```

I'm using Python version 3.8.5. I recommend having Python 3.6+ installed.

1.0.2 Comments

[8]:
```
# This is a single line comment. Comments allow you to write information within␣
↪your code that won't affect the behavior of your code.
```

[9]:
```
# Here's an example:

x = 1
# This comment won't break anything.
print(x)
x += 1
# Neither will this
print(x)
```

```
1
2
```

1.0.3 Datatypes and Operators

Python has a few primative datatypes which are important to know.

[76]:
```
# Integers
1
print(type(1))
```

```
# Floats
5.5
print(type(5.5))

# Booleans
True
False
print(type(True))

# Strings
"Hello World"
'Python Club'
print(type("Hello World"))

# Lists
[1, 2, 3]
print(type([1, 2, 3]))

# Tuples
(1, 2, 3)
print(type((1, 2, 3)))

# Dictionary
{'key1': 'value1', 'key2': 'value2'}
print(type({'key1': 'value1', 'key2': 'value2'}))
```

```
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
```

[26]:
```
# You can do math as expected
# Addition
1 + 1
```

[26]:
```
2
```

[28]:
```
# Subtraction
1 - 2
```

[28]:
```
-1
```

[27]:
```
# Multiplication
2 * 2
```

[27]: 4

[31]:
```
# Division
10 / 5
```

[31]: 2.0

[36]:
```
# Notice the output type of this division
type(10 / 5)
# Even though both numbers are integers, division will always return a float
```

[36]: float

[33]:
```
# To get an integer type back you have to use integer division, this will␣
↪always round the quotient down
10 // 5
```

[33]: 2

[35]:
```
type(10//5)
```

[35]: int

[37]:
```
print(24/5)
print(24//5)
```

```
4.8
4
```

[41]:
```
# However, if either of the numbers are a float, integer division will return a␣
↪float
24 // 5.0
# The result of / division is always a float
```

[41]: 4.0

[42]:
```
# Exponentiation
# Denoted as **
2**3
```

[42]: 8

[45]:
```
# PEMDAS
1 + 3 * 2
```

[45]: 7

[46]:
```
(1 + 3) * 2
```

[46]: 8

[47]:
```
# Booleans are written as True and False, but are actually 1 and 0
True + True
```

[47]: 2

[48]:
```
True - False
```

[48]: 1

[49]:
```
# Negation
not True
```

[49]: False

[50]:
```
# Boolean operators
True and True
```

[50]: True

[51]:
```
True and False
```

[51]: False

[52]:
```
True or False
```

[52]: True

[53]:
```
False or False
```

[53]: False

[54]:
```
# Equalities and Comparisons
1 == 1
```

[54]: True

[55]:
```
1 != 2
```

[55]: True

[56]:
```
1 < 2
```

[56]:
```
True
```

[57]:
```
1 > 2
```

[57]:
```
False
```

[3]:
```
# Modulo gets the remainder
10 % 5
```

[3]:
```
0
```

[4]:
```
10 % 4
```

[4]:
```
2
```

1.0.4 Printing

Printing allows you to output information to a console to see.

[11]:
```
print("Hello world")
print(1)
print(5.5)
```

```
Hello world
1
5.5
```

[21]:
```
# You can also print multiple things in one print statement
print("Hello", 2, "the", "World")
```

```
Hello 2 the World
```

[23]:
```
# When doing something like above, you can also specify how to seperate the␣
 ↪items being printed with the 'sep' parameter
print("Hello", 2, "the", "World", sep="*")
```

```
Hello*2*the*World
```

1.0.5 Variables

Python uses dynamic typing, meaning any varibale can be any datatype without being specified. This is different from other languages like Java or C++ which require the datatype to be defined at assignement.

[59]:
```
# Assign a varible using the equals sign
x = 1
print(x)
```

```
1
```

[62]:
```
x = 1
print(x)
x = "Hello"
print(x)
x = 5.5
print(x)
x = True
print(x)
x = []
print(x)
x = {}
print(x)
```

```
1
Hello
5.5
True
[]
{}
```

[61]:
```
# Python naming convention says to use underscores
my_var = 10
print(my_var)
```

```
10
```

[64]:
```
# Assigning multiple variables to multiple values
x, y, z = 1, 2, "Hello"
print(x)
print(y)
print(z)
```

```
1
2
Hello
```

[65]:
```
# Assigning multiple varibales to one value
x = y = z = "Hello World"
print(x)
print(y)
print(z)
```

```
Hello World
Hello World
Hello World
```

1.0.6 Lists, Tuples, and Dictionaries

[68]:
```
# Lists are defined with []
# Creates an empty list
x = []
print(x)
```

```
[]
```

[82]:
```
# Lists can contain data of any type, including other lists, called nested␣
↪lists. Values in a list are typically refered to as elements.
x = [1, 2.5, "Hello", ["World", "!"]]
print(x)
```

```
[1, 2.5, 'Hello', ['World', '!']]
```

[74]:
```
# Lists are subscriptable and start at index 0.
x = [1, 2.5, "Hello", ["World", "!"]]
print(x[0])
print(x[1])
print(x[2])
print(x[3])
print(x[3][0], x[3][1])
```

```
1
2.5
Hello
['World', '!']
World !
```

[75]:
```
# Lists can also be index backwards, starting from -1
x = [1, 2.5, "Hello", ["World", "!"]]
print(x[-1])
print(x[-2])
print(x[-3])
print(x[-4])
```

```
['World', '!']
Hello
2.5
1
```

[80]:
```
# You can get sub sets of lists by using subscripting and :
x = [1, 2.5, "Hello", ["World", "!"]]
print(x)

# The first number is the starting index, the second number is the ending index␣
↪which is not included in the subset.
```

```
# This subscript is saying get all the elments from 0 up to but not including 1
 ↪[0,1)
print(x[0:1])

# This is starting from index 2 to the end of the list
print(x[2:])

# From 2 to the end, but skip by 2 indicies
print(x[::2])

# From 2 down to but not including 0, skip by -1, meaning go backwards by 1
 ↪index
print(x[2:0:-1])
```

```
[1, 2.5, 'Hello', ['World', '!']]
[1]
['Hello', ['World', '!']]
[1, 'Hello']
['Hello', 2.5]
```

[81]:
```
# You can reassign values in a list by index. This is because lists are what
 ↪are described as mutable, meaning they can be changed.
x = [1, 2, 3]
print(x)
x[1] = 5
print(x)
```

```
[1, 2, 3]
[1, 5, 3]
```

[88]:
```
# Lists can also be manipulated through other built in functions
x = [1, 2, 3, 3]
print(x)

# Appends 4 to the back of the list
x.append(4)
print(x)

# Inserts a value into the list at a given index. In this case the index is 1
 ↪and the value is 5.
x.insert(1, 5)
print(x)

# Removes the first instance of the matching value from the list. The list has
 ↪two elements with the number 3, after using remove there is only one.
x.remove(3)
print(x)
```

```
# Remove an element at a specified index or from the back if no index is given. The element removed is also returned.
y = x.pop(1)
print(y)
print(x)

x.pop()
print(x)
```

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

[90]:
```
# Lists can be joined together.
x = [1, 2, 3]
y = [4, 5, 6]
z = x + y
print(z)

a = [1, 2, 3]
b = [4, 5, 6]
a.extend(b)
print(a)
```

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

[91]:
```
# Test if an element is in the list
x = [1, 2, 3]
print(1 in x)
print(4 in x)
```

```
True
False
```

[95]:
```
# You can find the length of the list, or how many elements are in it using len
x = [1, 2, 3]
y = len(x)
print(y)
```

```
3
```

[92]:
```
# Tuples are like lists but they are immutable, meaning they can't be changed␣
↪and edited like a list.
x = (1, 2, 3)
print(x[1])
```

```
2
```

[93]:
```
# Running this code will cause an error
x = (1, 2, 3)
x[1] = 4
print(x)
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-93-a0f59b6bfc4b> in <module>
  1 # Running this code will cause an error
  2 x = (1, 2, 3)
----> 3 x[1] = 4
  4 print(x)


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

[94]:
```
# You can cast the list type onto a tuple to convert it into a list
x = (1, 2, 3)
print(type(x))

y = list(x)
print(type(y))
```

```
<class 'tuple'>
<class 'list'>
```

[96]:
```
# Most of the list operations are still able to be used on a tuple
x = (1, 2, 3)
y = (4, 5, 6)
z = x + y
print(z)

print(z[3:])
```

```
print(len(z))

print(2 in z)
```

```
(1, 2, 3, 4, 5, 6)
(4, 5, 6)
6
True
```

[98]:
```
# Dictionaries store key to value pairs. You cam use keys to access the␣
↪information in a dictionary
x = {'key1':'value1', 'key2':'value2', 'key3':[1, 2, 3], 'key4':{}, 'key5':10}
print(x)
print(x['key1'])
print(x['key4'])
```

```
{'key1': 'value1', 'key2': 'value2', 'key3': [1, 2, 3], 'key4': {}, 'key5': 10}
value1
{}
```

[99]:
```
# You can get all the keys in a dictionary using the keys function
x = {'key1':'value1', 'key2':'value2', 'key3':[1, 2, 3], 'key4':{}, 'key5':10}
y = x.keys()
print(y)
```

```
dict_keys(['key1', 'key2', 'key3', 'key4', 'key5'])
```

[100]:
```
# Similarly, you can get just the values
x = {'key1':'value1', 'key2':'value2', 'key3':[1, 2, 3], 'key4':{}, 'key5':10}
y = x.values()
print(y)
```

```
dict_values(['value1', 'value2', [1, 2, 3], {}, 10])
```

[102]:
```
# You can also get both at once using the items function
x = {'key1':'value1', 'key2':'value2', 'key3':[1, 2, 3], 'key4':{}, 'key5':10}
y = x.items()
print(y)
```

```
dict_items([('key1', 'value1'), ('key2', 'value2'), ('key3', [1, 2, 3]),
('key4', {}), ('key5', 10)])
```

1.0.7 String Manipulation

[66]:
```
# Strings can be writted with "" or ''
x = "Hello"
y = 'World'
print(x, y)
```

```
Hello World
```

[103]:
```
# You can almost think of strings as a list of characters.
# For example: "Hey" is similar to ['H', 'e', 'y']
# That means strings can be manipulated similar to lists
x = "Hey"
print(x[0])
print(x[:2])
```

```
H
He
```

[104]:
```
# Like lists, strings can be added together, called concatenation
x = "Hello"
y = 'World'
z = x + " " + y
print(z)
```

```
Hello World
```

[105]:
```
# A very useful function is the split function, which seperates a string into a␣
 ↪list by a seperator character
x = "1,3,6,7"
y = x.split(',')
print(y)
```

```
['1', '3', '6', '7']
```

[108]:
```
# You can also use string formatting to maniupulate strings
a = 10

x = "There are " + str(a) + " bananas!"
print(x)

y = "There are {} apples!".format(a)
print(y)

z = f"There are {a} oranges!"
print(z)
```

```
There are 10 bananas!
There are 10 apples!
There are 10 oranges!
```

[109]:
```
# Strings can be multiplied, too
x = "***"
y = x * 10
print(y)
```

****

1.0.8 Functions

It is important to note the syntax moving forward. Python syntax is based on indentations. Anything indented is considered to be part of a block that begins with an indentation above it.

[110]:
```
# Functions are defined by the def keyword, followed by the function name and any parameters
# Note: the return statement is 1 indentation in meaning it is part of the function definition.
def square(x):
    return x * x

y = square(10)
print(y)
```

```
100
```

[111]:
```
def mult(x, y):
    return x * y

z = mult(5, 10)
print(z)
```

```
50
```

[112]:
```
# Functions don't always have to return a value

def my_func(x):
    print(f"It is {x} degrees today!")

my_func(75)
```

```
It is 75 degrees today!
```

1.0.9 Flow Control

[113]:
```
# If statements allow you to execute specific blocks of code if certain conditions exist

x = 10

if x > 0:
    print("x is positive")
elif x < 0:
    print("x is negative")
else:
```

```
    print("x is 0")
```

```
x is positive
```

[114]:
```
# The above code can also be put into a function

def pos_neg(x):
#   Again with indentation, anything even with this comment is part of the␣
 ↪function definition.
    if x > 0:
#       Anything even with this comment is considered part of it above if␣
 ↪statement block
        print("x is positive")
    elif x < 0:
        print("x is negative")
    else:
        print("x is 0")

pos_neg(-2)
pos_neg(0)
pos_neg(5)
```

```
x is negative
x is 0
x is positive
```

[116]:
```
# For loops allow you to go one item at a time through an iterable. Example of␣
 ↪iterables are lists, tuples, strings.
for i in [1, 2, 3]:
    print(i)
```

```
1
2
3
```

[117]:
```
x = "Hello world"
for i in x:
    print(i)
```

```
H
e
l
l
o

w
o
r
```

```
l
d
```

[119]:
```
# The range function is useful for creating an iterable that isn't equal to the value in a typical iterable.
# The range function works as follows: range(start, stop, step). The default step is 1
x = range(0, 10)
for i in x:
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

[120]:
```
x = range(0, 10, 2)
for i in x:
    print(i)
```

```
0
2
4
6
8
```

[122]:
```
x = range(10, 0, -2)
for i in x:
    print(i)
```

```
10
8
6
4
2
```

[123]:
```
# A popular use of range is to use it with the length of list, string, tuple, or another type that has subscripting
x = [10, 5, 4]
for i in range(len(x)):
    print(i)
```

```
0
1
2
```

[1]:
```
# While loops allow a block of code to run until a condition is met. While␣
↪loops will test the condition first, meaning the loop may never run.
x = 1
while x < 10:
    print(x)
    x += 1
```

```
1
2
3
4
5
6
7
8
9
```

[1]:
```
# Continue and Break keywords allow you to stop or skip an iteration inside a␣
↪for or while loop

for i in range(10):
#   if i is even, go to the next iteration
    if i % 2 == 0:
        continue
    else:
        print(i)
```

```
1
3
5
7
9
```

[2]:
```
# This while loop will run forever until you provide the input with a valid␣
↪response. In this case the valid response is anything.
while True:
    x = input("What is your name?")
    if x != "":
        print("Hello " + x)
        break
```

```
What is your name?
What is your name?
What is your name?
```

```
What is your name? Scott
Hello Scott
```

[ ]: