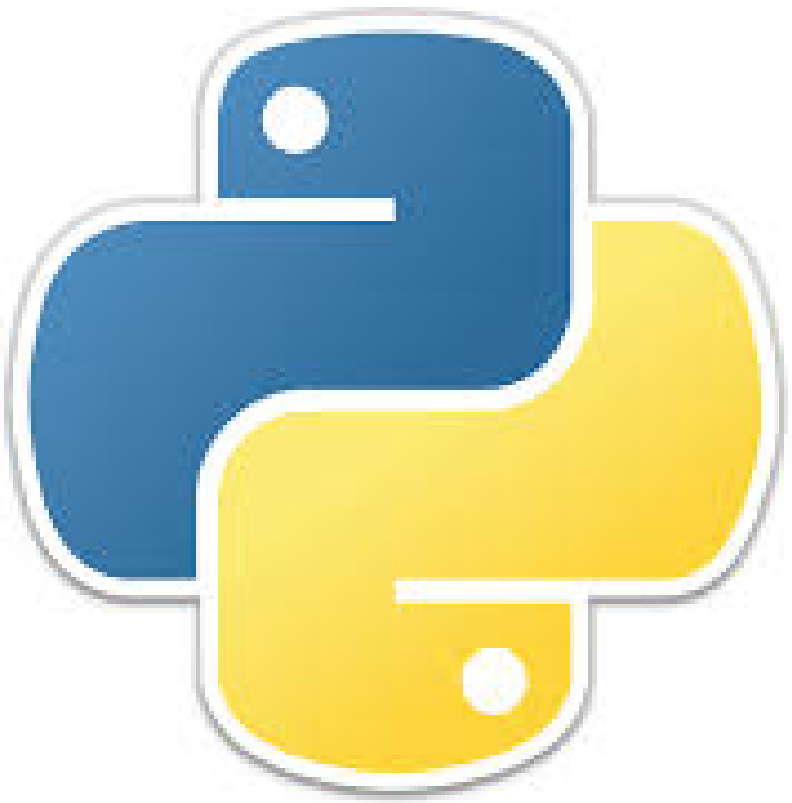

Python Basics

20.09.2019

Python Notes

INTRO

What is Python?

- Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

Good to know

- Python has a simple syntax similar to the English language.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

SYNTAX

Identation

- Indentation refers to the spaces at the beginning of a code line.
- Python uses indentation to indicate a block of code.

```
Example:    CORRECT
```

```
if 5 > 2:
   print("Five is greater than two!")
```

Example: SYNTAX ERROR

```
if 5 > 2:
print("Five is greater than two!")
```

Comments

Example:

```
x = 10 # this is a comment
"""
This can be used for multi-line comments
"""
```

VARIABLES

Basics

- In Python variables are created the moment you assign a value to it.
- Python has no command for declaring a variable.

Example:

```
x = 5 # type of int
y = "Hello, World!" # type of str

#String variables can be declared either by using single or double quotes:
z = 'Hey' # type of str

#  Assign values to multiple variables in one line:
a, b, c = "Orange", "Banana", "Cherry"

# Python print statement is often used to output variables.
x = "awesome"
print("Python is " + x)
```

Global Variables

- Variables that are created outside of a function (as in all of the examples above) are known as global variables.
- Global variables can be used by everyone, both inside of functions and outside.

Example: Create a variable outside of a function, and use it inside the function

```
x = "awesome"

def myfunc():
   print("Python is " + x)

myfunc()
```

- If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

Example: Create a variable inside a function, with the same name as the global variable

```
x = "awesome"

def myfunc():
  x = "fantastic"
  print("Python is " + x) # prints Python is fantastic

myfunc()
print("Python is " + x)  # prints Python is awesome
```

- Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.
- To create a global variable inside a function, you can use the **`global`** keyword.

Example: If you use the `global` keyword, the variable belongs to the global scope:

```
def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x) # global variable x is still reachable.
```

- Use the **`global`** keyword if you want to change a global variable inside a function.

Example: To change the value of a global variable inside a function, refer to the variable by using the `global` keyword:

```
x = "awesome"

def myfunc():
  global x
  x = "fantastic" # now the global variable x has changed as fantastic!

myfunc()

print("Python is " + x) # prints Python is fantastic.
```

DATA TYPES

Built-in Data Types

- Variables can store data of different types, and different types can do different things.
- Python has the following data types built-in by default, in these categories:

```
Text Type:      str
Numeric Types:  int, float, complex
Sequence Types: list, tuple, range
```

```
Mapping Type:   dict
Set Types:      set, frozenset
Boolean Type:   bool
Binary Types:   bytes, bytearray, memoryview
```

Getting the Data Type

- You can get the data type of any object by using the **`type()`** function:

Example: Print the data type of the variable x:

```
x = 5
print(type(x)) #prints int
```

Setting the Data Type

- In Python, the data type is set when you assign a value to a variable:

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

NUMBERS

Basic Numeric Types

- There are three numeric types in Python:
  - int
  - float
  - complex

Example:

```
x = 5
y = 2.8
z = 3+5j

#To verify the type of any object in Python, use the type() function:
```

```
print(type(x))
print(type(y))
print(type(z))
```

Random Number

- Python has a built-in module called `random` that can be used to make random numbers:

Example: Import the random module, and display a random number between 1 and 9:

```
import random
print(random.randrange(1,10))
```

CASTING

- There may be times when you want to specify a type on to a variable. This can be done with casting
- Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types
- Casting in python is therefore done using constructor functions:
  - **`int()`**
  - **`float()`**
  - **`str()`**

Example: Integers

```
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

Example: Floats

```
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2
```

Example: Strings

```
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

STRINGS

- `'hello'` is the same as `"hello"`.
- You can assign a multiline string to a variable by using `""" """`:

Strings are Arrays

- Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.
- However, Python does not have a character data type, a single character is simply a string with a length of 1.
- Square brackets can be used to access elements of the string.

Example: Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1])
```

Slicing

Example: Get the characters from position 2 to position 5 (not included):

```
b = "Hello, World!"
print(b[2:5]) # prints llo
```

Some Built-in String Methods

- Python has a set of built-in methods that you can use on strings.

Examples:

#strip() method removes any whitespace from the beginning or the end
#lower() method returns the string in lower case
#upper() method returns the string in upper case

#replace() method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

#split() method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

Check String

- To check if a certain phrase or character is present in a string, we can use the keywords **`in`** or **`not in`**.

Example: Check if the phrase "ain" is present in the following text:

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" in txt
print(x) #prints true
```

String Format

- As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

Example:

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

```
TypeError: must be str, not int
```

- But we can combine strings and numbers by using the `format()` method!
- The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders `{}` are
- The format() method takes unlimited number of arguments, and are placed into the respective placeholders

Example: Use the format() method to insert numbers into strings:

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

BOOLEANS

- Booleans represent one of two values: `True` or `False`.
- The `bool()` function allows you to evaluate any value, and give you `True` or `False` in return,

Example:

```
x = "Hello"
y = 15
bool(x) #returns true
bool(y) #returns true

#below ones returns false
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```

Most Values are True

- Any string is `True`, **except empty strings**.
- Any number is `True`, **except 0**.
- Any list, tuple, set, and dictionary are `True`, **except empty ones.**

OPERATORS

- Operators are used to perform operations on variables and values.
- Python divides the operators in the following groups:
  - Arithmetic operators
  - Assignment operators
  - Comparison operators
  - Logical operators
  - Identity operators
  - Membership operators
  - Bitwise operators

Arithmetic Operators

- Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	`x + y`
-	Subtraction	`x - y`
*	Multiplication	`x * y`
/	Division	`x / y`
%	Modulus	`x % y`
**	Exponentiation	`x ** y`
//	Floor division	`x // y`

Assignment Operators

- Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	`x = 5`
+=	x += 3	`x = x + 3`
-=	x -= 3	`x = x - 3`
*=	x *= 3	`x = x * 3`
/=	x /= 3	`x = x / 3`
%=	x %= 3	`x = x % 3`
//=	x //= 3	`x = x // 3`
**=	x **= 3	`x = x ** 3`
&=	x &= 3	`x = x & 3`
|=	x |= 3	`x = x | 3`
^=	x ^= 3	`x = x ^ 3`
>>=	x >>= 3	`x = x >> 3`
<<=	x <<= 3	`x = x << 3`

Comparison Operators

- Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	`x == y`
!=	Not equal	`x != y`
>	Greater than	`x > y`
<	Less than	`x < y`
>=	Greater than or equal to	`x >= y`
<=	Less than or equal to	`x <= y`

Logical Operators

- Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	`x < 5 and  x < 10`
or	Returns True if one of the statements is true	`x < 5 or x < 4`
not	Reverse the result, returns False if the result is true	`not(x < 5 and x < 10)`

Identity Operators

- Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns true if both variables are the same object	`x is y`

is not	Returns true if both variables are not the same object	`x is not y`

Membership Operators

- Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	`x in y`
not in	Returns True if a sequence with the specified value is not present in the object	`x not in y`

Bitwise Operators

- Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

SCOPE

Intro

- A variable is only available from inside the region it is created. This is called **scope**.

Local Scope

- A variable created inside a function belongs to the ***local scope*** of that function, and can only be used inside that function.

Example: A variable created inside a function is not available outside the function, but it is available inside that function or for any function inside the function:

```
def myfunc():
  x = 300 # local variable
  def myinnerfunc():
    print(x) # x is available here

# x is not available here
```

Global Scope

- A variable created in the main body of the Python code is a global variable and belongs to the global scope.
- Global variables are available from within any scope, global and local.

Example: A variable created outside of a function is global and can be used by anyone:

```
x = 300 # global variable

def myfunc():
```

```
  print(x)

myfunc()
print(x)
```

Naming Variables

- If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):

Example: The function will print the local x, and then the code will print the global x:

```
x = 1 # global

def myfunc():
  x = 2 # local
  print(x)

myfunc() # prints local variable 2
print(x) # prints global variable 1
```

Global Keyword

- If you need to create a global variable, but are stuck in the local scope, you can use the `global` keyword.
- The `global` keyword makes the variable global.

Example: If you use the global keyword, the variable belongs to the global scope:

```
def myfunc():
  global x
  x = 300

myfunc()
print(x) # x is a global variable
```

- Also, use the **`global`** keyword if you want to make a change to a global variable inside a function.

Example: To change the value of a global variable inside a function, refer to the variable by using the global keyword:

```
x = 1

def myfunc():
  global x
  x = 2

myfunc()
print(x) # prints 2
```

EXCEPTION HANDLING

Intro

- The **`try`** block lets you test a block of code for errors.
- The **`except`** block lets you handle the error.

- The `finally` block lets you execute code, regardless of the result of the rty- and except blocks.

Exception Handling

- When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
- These exceptions can be handled using the `try` statement:

Example: The try block will generate an exception, because x is not defined:

```
try:
  print(x) # since x is not defined the try block will generate an exception.
except:
  print("An exception occurred")
```

- Since the try block raises an error, the except block will be executed.
- Without the try block, the program would crash and raise an error:

Many Exceptions

- You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:
- You can use the `else` keyword to define a block of code to be executed if no errors were raised:

Example: Print one message if the try block raises a NameError and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
else:
  print("Nothing went wrong")
```

Finally

- The `finally` block, if specified, will be executed regardless if the try block raises an error or not.
- This can be useful to close objects and clean up resources:

Example: Try to open and write to a file that is not writable:

```
try:
  f = open("demofile.txt")
  f.write("Lorum Ipsum")
except:
  print("Something went wrong when writing to the file")
finally:
  f.close() # f will be closed in all conditions
```

More blogs