Python Basics

Course 4- Coursera Week 3

Objectives

- Understand Python Conditions and Branching
- Understand Python Loops
- Understand Python Functions
- Understand Python Objects and Classes

Conditions

In Python, conditions are similar to all of the C-like languages. We write conditions using the `if` keyword, which is followed by a logical expression and then by a colon (`:`). If the expression is true, the following statement will be executed. If it's not true, the following statement will be skipped, and the program will continue with the next statement.

Try it

```
#!/usr/bin/env python3
if 15 > 5:
print("True")
print("The program continues here...")
```

Let's look at the relational operators which can be used in expressions:

Meaning	Operator
Equal to	==
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=
Not equal	!=
Negation	not

We use the **==** operator for equality to avoid confusing it with a normal assignment to a variable **(the = operator)**. If we want to negate an expression, we write it in parentheses using the negation operator **(!)** before the actual expression within the parentheses. If you want to execute more than one command, you have to indent each line with a tab:

The program retrieves a number from the user, and it calculates its square root (if it is greater than 0). We have used the **** operator and set the variable a to be computed with an exponent of **1/2**, which is the equivalent to getting its square root.

Try it

```
a = int(input("Enter a number and I'll get its square root: "))
if (a > 0):
print("The number you entered is greater than 0, so I can calculate it!")
root = a ** (1/2)
print("The square root of %d is %f" % (a, root))
if (a <= 0):
print("I can't calculate the square root of a negative number!")
print("Thanks for the input!")
```

Conditions can be composed using two basic logical operators:

Operator	**syntax**
Logical AND	and
Logical OR	or

Python Loops

Python has only two loops:

1. for loop
2. while loop

For loop

The for loop Syntax:

```
for i in iterable_object:
    # do something
```

Note:

All the statements inside for and while loop must be indented to the same number of spaces. Otherwise, **SyntaxError** will be thrown.

Try it

```python
my_list = [1,2,3,4]
for i in my_list:
 print(i)
```

range(a, b) Function

The **range(a, b)** functions returns sequence of integers from **a**, **a + 1**, **a+ 2** .... , **b - 2**, **b - 1**.

Try it

```
for i in range(1, 10):
    print(i)
```

Another way, Try it

You can also use the **range()** function by supplying only one argument like this:

```
for i in range(10):
    print(i)
```

The **range(a, b)** function has an optional third parameter to specify the step size.

Try it

```
for i in range(1, 20, 2):
print(i)
```

While loop

Syntax:

```
while condition:
    # do something
```

The while loop keeps executing statements inside it until condition becomes false. After each iteration condition is checked and if it's True then once again statements inside the while loop will be executed.

Try it

```
count = 0
while count < 10:
print(count)
count += 1
```

Here while will keep printing until **count** is less than **10**

BREAK STATEMENT

The **break** statement allows to breakout out of the loop.

[Try it]

```
count = 0
while count < 10:
count += 1
 if count == 5:
break
print("inside loop", count)
print("out of while loop")
```

when **count** equals to **5** if condition evaluates to **True** and **break** keyword breaks out of loop.

CONTINUE STATEMENT

When **continue** statement encountered in the loop, it ends the current iteration and programs control goes to the end of the loop body.

Try it

```
count = 0
while count < 10:
count += 1
if count % 2 == 0:
continue
print(count)
```

As you can see, when **count % 2 == 0**, the **continue** statement is executed which causes the current iteration to end and the control moves on to the next iteration.

Python Functions

Functions are the reusable pieces of code which helps us to organize structure of the code. We create functions so that we can run a set of statements multiple times during in the program without repeating ourselves.

Creating functions

Python uses def keyword to start a function, here is the syntax:

```
def function_name(arg1, arg2, arg3, .... argN):

     #statement inside function
```

Note:

All the statements inside the function should be indented using equal spaces. Function can accept zero or more arguments(also known as parameters) enclosed in parentheses. You can also omit the body of the function using the **pass** keyword, like this:

```
def myfunc():

    pass
```

Try it

```
def sum(start, end):
result = 0
 for i in range(start, end + 1):
 result += i
print(result)
sum(10, 50)
```

Function with return value

The above function simply prints the result to the console, what if we want to assign the result to a variable for further processing? Then we need to use the **return** statement. The **return** statement sends a result back to the caller and exits the function.

Try it

```python
def sum(start, end):
result = 0
for i in range(start, end + 1):
result += i
return result
s = sum(10, 50)
print(s)
```

Global variables vs local variables

Global variables: Variables that are not bound to any function , but can be accessed inside as well as outside the function are called global variables.

Local variables: Variables which are declared inside a function are called local variables.

Try it

```
global_var = 12        # a global variable
def func():
local_var = 100     # this is local variable
print(global_var)   # you can access global variables in side function
func()                 # calling function func()
#print(local_var)       # you can't access local_var outside the function, because as soon as function ends
local_var is destroyed
```

You can bind local variable in the global scope by using the **global** keyword followed by the names of variables separated by comma **(,).**

Try it

```
t = 1
def increment():
global t   # now t inside the function is same as t outside the function
t = t + 1
print(t) # Displays 2
increment()
print(t) # Displays 2
```

Argument with default values

To specify default values of argument, you just need to assign a value using assignment operator.

```
def func(i, j = 100):
    print(i, j)
```

Above function has two parameter i and j. The parameter j has a default value of 100, it means that we can omit value of j while calling the function.

Try it

```
func(2) # here no value is passed to j, so default value will be used
```

Call the func() function again, but this time provide a value to the j parameter.

Try it

```
func(2, 300) # here 300 is passed as a value of j, so default value will not be used
```

Keyword arguments

There are two ways to pass arguments to method: positional arguments and Keyword arguments.

We have already seen how positional arguments work in the previous section. In this section we will learn about keyword arguments.

Keyword arguments allows you to pass each arguments using name value pairs like this name=value. Let's take an example:

```
def named_args(name, greeting):

    print(greeting + " " + name )
```

Try it

```
named_args(name='jim', greeting='Hello')
named_args(greeting='Hello', name='jim') # you can pass arguments this way too
```

Mixing Positional and Keyword Arguments

It is possible to mix positional arguments and Keyword arguments, but for this positional argument must appear before any Keyword arguments.

```
def my_func(a, b, c):
    print(a, b, c)
```

Try it

1	def my_func(a , b , c):
2	print(a,b,c)

3	# using positional arguments only
	my_func(12, 13, 14)

4	# here first argument is passed as positional arguments while other two as keyword argument
	my_func(12, b=13, c=14)

5	# same as above
	my_func(12, c=13, b=14)

6	# this is wrong as positional argument must appear before any keyword argument # my_func(12, b=13, 14)

Returning multiple values from Function

We can return multiple values from function using the return statement by separating them with a comma (,). Multiple values are returned as tuples.

Try it

```
def bigger(a, b):
    if a > b:
        return a, b
 else:
        return b, a
s = bigger(12, 100)
print(s)
print(type(s))
```

Python Object and Classes

Python is an object-oriented language. In python everything is object i.e **`int`**, **`str`**, **`bool`** even modules, functions are also objects.

Object oriented programming use objects to create programs, and these objects stores data and behaviours.

Defining class

Class name in python is preceded with **class** keyword followed by a colon (**:**). Classes commonly contains data field to store the data and methods for defining behaviors. Also every class in python contains a special method called *initializer* (also commonly known as constructors), which get invoked automatically every time new object is created.

```
class Person:
# constructor or initializer
        def __init__(self, name):
self.name = name # name is data field also commonly known as instance variables
# method which returns a string
       def whoami(self):
 return "You are " + self.name
```

Here we have created a class called **Person** which contains one data field called **name** and method **whoami()**.

What is self?

All methods in python including some special methods like initializer have first parameter **self**. This parameter refers to the object which invokes the method. When you create new object the **self** parameter in the **__init__** method is automatically set to reference the object you have just created.

Creating object from class

Try it

```python
p1 = Person('tom') # now we have created a new person object p1
print(p1.whoami())
print(p1.name)
```

Note:

When you call a method you don't need to pass anything **to `self`** parameter, python automatically does that for you behind the scenes.

Hiding data fields

To hide data fields you need to define private data fields. In python you can create private data field using two leading underscores. You can also define a private method using two leading underscores.

1	class BankAccount:
2	# constructor or initializer def __init__(self, name, money):
3	self.__name = name
4	self.__balance = money # __balance is private now, so it is only accessible inside the class
5	def deposit(self, money):
6	self.__balance += money
7	def withdraw(self, money):
8	if self.__balance > money :

9	self.__balance -= money
	return money
	else:
	return "Insufficient funds"

10	def checkbalance(self):
	return self.__balance
	
	b1 = BankAccount('tim', 400)
	print(b1.withdraw(500))
	b1.deposit(500)
	print(b1.checkbalance())
	print(b1.withdraw(800))