# **Programming with Python**

# Module 6:



* Abstraction and encapsulation
* Functions
* Classes and Methods

## Abstraction and encapsulation

Abstraction is one of the 3 Key Principles of Object Oriented Programming. Encapsulation is a mechanism you use to provide abstraction.

#-- Data --#  
# declare variables and constants  
fltN1 **=** 0.0  
fltN2 **=**0.0  
  
#-- Processing --#  
# perform tasks  
**def** DivideValues**():** print**(**fltN1 **/** fltN2**) #I have to know what code is in the function**  
  
#-- Presentation (Input/Output) --#  
# get user input  
fltN1 **=** float**(**input**("Enter the first number: "))#I dont need to know**fltN2 **=** float**(**input**("Enter the second number: "))#the code in input()**# send program output  
DivideValues**()**

**NOTE: Try to organize your code into three sections: Data, Processing, and Presentation.**

Having a function that prints data to a user is not considered a good choice since it mixes processing code with presentation code. A better way to write the code is to return a value from the function and have separate code for presentation.

## Functions

Functions are a way of grouping one or more statements together. You can then execute these statements by "calling" the function.

# Define the function

def ProcessSomething():

print("I'm") # first statement

print("processing data")# second statement

# Call the function

ProcessSomething()

Note: You have to define the function before the code that calls the function.

### Parameters

Optionally, function can have parameters. These allow you to pass in values for processing.

# Define the function

def ProcessSomething(parmMessage):

print("The parameter was: " + parmMessage)

# Call the function

ProcessSomething("arg ABC")

There is no practical limit to how many parameters you include.

# Define the function

def AddValues(value1, value2):

decAnswer = value1 + value2

print("The Sum of the values is: " + str(decAnswer))

# Call the function

AddValues(10, 5)



**LAB 6-1: Working with functions**

1. Create a function that prints the Sum, Difference, Product, and Quotient of two numbers.



### Return values

Functions can return values and evaluate as expressions.

# Define the function

def AddValues(value1, value2):

decAnswer = value1 + value2

**return decAnswer**

# Call the function

print("The Sum of the values is: " + str(AddValues(10, 5)))

Return values can be a single item of data or multiple items.

**# -- data code --**

v1 = None # first argument

v2 = None # second argument

decAnswer = None # result of processing

**# --processing code--**

# Define the function

def AddValues(value1, value2):

decAnswer = value1 + value2

**return decAnswer, value1, value2**

**# --presentation (I/0) code--**

# Call the function

**decAns, v1, v2** = AddValues(10, 5)

print("The Sum of the values "

+ str(v1) + " and " + str(v2) + " is: " + str(decAns) )

Most languages do not let you return multiple values in this manner. Instead you would return a collection of values in a tuple(array) or list. Here is an example using a list instead.

# -- data code --

v1 = None # first argument

v2 = None # second argument

lstData = None # result of processing

# --processing code--

# Define the function

def AddValues(value1, value2):

decAnswer = value1 + value2

**lstResults = [decAnswer, value1, value2]**

**return lstResults**

# --presentation (I/0) code--

# Call the function

v1 = 10

v2 = 5

lstData = AddValues(v1, v2)

print("The Sum of the values "

+ str(lstData[1]) + " and " + str(lstData[2]) + " is: " + str(lstData[0]) )



**LAB 6-2: Working with returned lists**

1. Create a function that returns a list of the Sum, Difference, Product, and Quotient of two numbers.
2. Display the results to the user.
3. Divide you program into data, processing, and presentation sections.



### Positional vs Named arguments

When you call a function you can include the name of the parameter and fill it explicitly.

# -- data code --

v1 = 0 # first argument

v2 = 0 # second argument

decAns = 0 # result of processing

# --processing code--

# Define the function

def AddValues(value1, value2):

decAnswer = value1 + value2

lstResults = [decAnswer, value1, value2]

return lstResults

# --presentation (I/0) code--

# Call the function

lstData = AddValues**(value1 = 10, value2 = 5)**

print("The Sum of the values "

+ str(lstData[1]) + " and " + str(lstData[2]) + " is: " + str(lstData[0]) )

### Default Parameter Values

You can set default values for a parameter if you want to.

# -- data code --

v1 = None # first argument

v2 = None # second argument

lstData = None # result of processing

# --processing code--

# Define the function

def AddValues(**value1 = 0, value2 = 0**):

decAnswer = value1 + value2

lstResults = [decAnswer, value1, value2]

return lstResults

# --presentation (I/0) code--

# Call the function

lstData = **AddValues(10)**

print(lstData)

lstData = **AddValues(value1 = 10)**

print(lstData)

lstData = **AddValues(value2 = 5)**

print(lstData)

### Overloading

Most OOP languages allow for multiple versions of a function to be included in your code, but Python does not support this in the traditional sense. This missing technique is called function overloading.

# -- data code --

v1 = None # first argument

v2 = None # second argument

lstData = None # result of processing

# --processing code--

# Define the function

def AddValues(value1, value2):

decAnswer = value1 + value2

lstResults = [decAnswer, value1, value2]

return lstResults

**# This would overwrite the original function *(in other languages!)***

**def AddValues(value1, value2, value3):**

decAnswer = value1 + value2 + value3

lstResults = [decAnswer, value1, value2, value3]

return lstResults

# --presentation (I/0) code--

# Call the function

lstData = **AddValues(5,10)** ***#TypeError: AddValues() takes exactly 3 arguments***

print(lstData)

lstData = AddValues(5,10,15)

print(lstData)

However, you can use default parameters values to mimic this behavior.

# -- data code --

v1 = None # first argument

v2 = None # second argument

lstData = None # result of processing

# --processing code--

# Define the function

def AddValues(**value1 = None, value2 = None, value3 = None**):

**lstResults = None #Holds results list**

**if (value3 is None):**

**decAnswer = value1 + value2**

**lstResults = [decAnswer, value1, value2]**

**else:**

**decAnswer = value1 + value2 + value3**

**lstResults = [decAnswer, value1, value2, value3]**

**return lstResults**

# --presentation (I/0) code--

# Call the function

lstData = AddValues(5,10)

print(lstData)

lstData = AddValues(5,10,15)

print(lstData)

“**None**

The sole value of the type NoneType. None is frequently used to represent the absence of a value, as when default arguments are not passed to a function. Assignments to None are illegal and raise a [SyntaxError](https://docs.python.org/3.5/library/exceptions.html#SyntaxError).” (Python docs, <https://docs.python.org/3.5/library/constants.html>, 2015)

### Returning Data by Reference

In most languages you can pass in arguments as a reference to its address in memory, change the value of the referenced memory space, and then use that value outside of the function. Unfortunately, Python does not support this functionality, at least not for simple data types like strings and integers.

# -- data code --

v1 = 10 # first argument

v2 = 5 # second argument

**decAnswer = None; # result of processing**

# --processing code--

# Define the function

def AddValues(value1, value2, **answer**):

**#answer will receive the value not the address!**

**answer** = value1 + value2

# --presentation (I/0) code--

# Call the function

AddValues(v1, v2, **decAnswer**)

print("The Sum of the values "

+ str(v1) + " and " + str(v2) + " is: " + str(decAnswer) )

*# This prints out -- "The Sum of the values 10 and 5 is: None." Doh!*

However, one way around this issue is to pass in a List object. Here is an example.

# -- data code --

v1 = 10 # first argument

v2 = 5 # second argument

**decAnswer** = **[None]** # result of processing using a list

# --processing code--

# Define the function

def AddValues(value1, value2, answer):

**#answer receives the address this time since list act as ref types!**

**answer[0]** = value1 + value2

**#answer is a list object, so you use an index [0]!**

# --presentation (I/0) code--

# Call the function

AddValues(v1, v2, **decAnswer**)

print("The Sum of the values "

+ str(v1) + " and " + str(v2) + " is: " + str(**decAnswer[0]**) )

# This prints out -- "The Sum of the values 10 and 5 is: **15**"

### Global Variables

Another way to change the values of variables declared outside of the function, within a function, is to use a "global" variable.

Note: This practice is discouraged since it breaks the concept of "Encapsulation/Abstraction."

# -- data code --

# Note: Variables declared in the body of the script are called "Global"

v1 = 10 # first argument

v2 = 5 # second argument

**decAnswer = None # result of processing**

# --processing code--

# Define the function

def AddValues(value1, value2):

**decAnswer = 0 # This is a "local variable!"**

**global decAnswer # decAnswer now refers to the "global variable"**

**answer = value1 + value2**

**decAnswer = value1 + value2 # This refers to a local variable**

# --presentation (I/0) code--

# Call the function

AddValues(v1, v2)

print("The Sum of the values "

+ str(v1) + " and " + str(v2) + " is: " + str(**decAnswer**) )

# This prints out -- "The Sum of the values 10 and 5 is: **15**"

### Shadowing a global variable

Be careful to always use the keyword *global* if you use this technique or your local variable will shadow the global one!

# -- data code --

# Note: Variables declared in the body of the script are called "Global"

v1 = 10 # first argument

v2 = 5 # second argument

decAnswer = None # result of processing

print(globals()) #This function shows what global variables are running

# --processing code--

# Define the function

def AddValues(value1, value2):

answer = 0 # This is a "local variable!"

answer = value1 + value2

**decAnswer = answer**

**# decAnswer does NOT refer to the "global variable, but instead**

**# "shadows" it!**

# --presentation (I/0) code--

# Call the function

AddValues(v1, v2)

print("The Sum of the values "

+ str(v1) + " and " + str(v2) + " is: " + str(**decAnswer**) )

# This prints out -- "The Sum of the values 10 and 5 is: **None**"

“**What are the rules for local and global variables in Python?**

In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a value anywhere within the function’s body, it’s assumed to be a local unless explicitly declared as global.

Though a bit surprising at first, a moment’s consideration explains this. On one hand, requiring global for assigned variables provides a bar against unintended side-effects. On the other hand, if global was required for all global references, you’d be using global all the time. You’d have to declare as global every reference to a built-in function or to a component of an imported module. This clutter would defeat the usefulness of the global declaration for identifying side-effects.”

(Python Docs, https://docs.python.org/3.5/faq/programming.html, 2015)

### Function Document Headers (Doc Strings)

It is a common practice to include a header at the beginning of a function. This is known as *docstring* in python.

def AddValues(value1, value2):

**""" This function adds two values """**

return v1 + v2

Something your employer will want you to include a more description doc string. Here is an example:

**def** AddValues**(**value1**,** value2**):** """   
 :**Desc** : This function adds two values   
 **:param** value1: the first number to add  
 **:param** value2: the second number to add  
 **:type** value1: decimal or int  
 **:type** value2: decimal or int  
 **:return**: returns a sum of two numbers  
 **:rtype**: decimal or int  
 """  
 **return** value1 **+** value2  
  
print**(**AddValues**(**4**,**5**))**

## Classes and Methods

Classes are a way of grouping functions, variables, and constants. You can make a class with the following syntax.

Note that the function is now called a Method.

# -- data code --

v1 = 10 # first argument

v2 = 5 # second argument

decAnswer = None # result of processing

# -- processing code --

# Create a Class to hold a list of functions

class MathProcessor(object):

""" This class contains methods for processing simple math """

# Define the method

@staticmethod #We will talk about this in Module 7

def AddValues(value1, value2):

""" This function adds two values """

decAnswer = value1 + value2

return decAnswer

# -- presentation (I/0) code --

# Call the method (function)

decAnswer = MathProcessor.AddValues(v1, v2)

print("The Sum of the values "

+ str(v1) + " and " + str(v2) + " is: " + str(decAnswer))

# This prints out -- "The Sum of the values 10 and 5 is: 15"

Classes can group data as well as methods. Here is an example.

#--- Make the class ---

class Customer(object):

Id = None

Name = None

def ToString(**self**):

return str(self.Id) + "," + str(self.Name)

# --- Use the class ----

objCustomer1 = Customer() #This code creates a “Copy” of the class

objCustomer1.Id = 1

objCustomer1.Name = "Bob Smith"

objCustomer2 = Customer() #This code creates different “Copy”

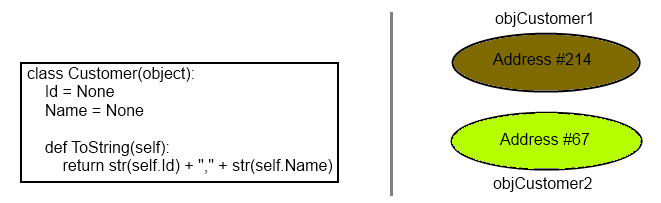
objCustomer2.Id = 2

objCustomer2.Name = "Sue Jones"

print(objCustomer1.ToString())

print(objCustomer2.ToString())

In this example the method includes the keyword ***self*** as the first parameter, but other than that they act just like functions. T*he keyword* self *refers to the objects address space.*





**LAB 6-3: Working with classes**

1. Create a class with (4) Methods, have them return a Sum, Difference, Product, and Quotient of two numbers.
2. Name the class MathProcessor
3. Name the methods **AddValues, SubtractValues, MultiplyValues, DivideValues.**
4. Display the results to the user by calling each method.

