

Ch03-3-Functions-UserDefined

August 7, 2020

1 3.3 User-defined Functions

- <http://openbookproject.net/thinkcs/python/english3e/functions.html>

1.1 Topics

- how to define and use your own functions
- variables scopes
- how to pass data to functions (by value and reference)
- how to return data from functions
- unit or functional testing with assert

1.2 3.3.1 Functions

- named sequence of statements that execute together to solve some task
- primary purpose is to help us break the problem into smaller sub-problems or tasks
- two types: fruitful and void/fruitless functions
- must be defined before it can be used or called (two step process)
- concept of function is borrowed from Algebra
- e.g.

Let's say: $y = f(x) = x^2 + x + 1$

$y = f(1) = 1 + 1 + 1 = 3$

$y = f(-2) = 4 - 2 + 1 = 3$

1.2.1 Two-step process

1. Define a function
2. Call or use function

1.2.2 syntax to define function

```
def functionName( PARAMETER1, PARAMETER2, ... ):
    # STATEMENTS
    return VALUE
```

- PARAMETERS and return statements are OPTIONAL

- function NAME follows the same rules as a variable/identifier name
- recall some built-in functions and object methods have been used in previous chapters...

1.2.3 syntax to call function

- call function by its name
- use return values if any

VARIABLE = functionName(ARGUMENT1, ARGUMENT2, ...)

1.3 3.3.2 Why functions?

dividing a program into functions or sub-programs have several advantages: - give you an opportunity to name a group of statements, which makes your program easier to read and debug - can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place - allow you to debug the parts one at a time (in a team) and then assemble them into a working whole - write once, test, share, and reuse many times (libraries, e.g.)

1.4 3.3.3 Types of functions

- two types: fruitful and fruitless functions

1.5 Fruitless functions

- also called void functions
- they do not return a value

```
[4]: # Function definition
      # function prints the result but doesn't explicitly return anything
      def greet():
          print('Hello World!')
```

```
[5]: # Function call
      greet()
      greet()
```

Hello World!
Hello World!

```
[6]: # void/fruitless function; returns None by default
      a = greet() # returned value by greet() assigned to a
      print('a =', a)
```

Hello World!
a = None

```
[4]: type(greet)
```

```
[4]: function
```

```
[5]: # function can be assigned to another identifier
myfunc = greet
type(myfunc)
```

```
[5]: function
```

```
[6]: myfunc()
```

Hello World!

1.6 Fruitful functions

- functions that explicitly return some value(s) using **return** statement
- more useful functions
- answer returned can be used as intermediate values to solve bigger problems
- can be used and tested independently
- fruitful functions usually take some arguments and return value(s) as answer
- most built-in and library functions are fruitful
- typically return is the last statement to execute; but not necessarily
- function returns back to the caller immediately after return statement is executed
 - will skip code if any exists after return statement

```
[3]: # fruitful function
def getName():
    name = input("Hi there, enter your full name: ")
    return name
    print(f'Hi {name}, nice meeting you!') # will not be executed
```

```
[2]: userName = getName()
```

Hi there, enter your full name: John Smith

1.7 3.3.4 Passing data as arguments to functions

- functions are subprograms that may need external data to work with
- you can pass data to functions via parameters/arguments
- can provide 1 or more parameters to pass 1 or more data
- can provide default values to parameters
 - makes the parameter optional when the function is called
- if a function has a required parameter, data must be provided for each required parameter!
 - otherwise, you'll get error!

1.7.1 Visualize with PythonTutor.com

```
[7]: # function takes one required argument
def greet(name):
    print(f'Hello {name}')
```

```
[8]: # pass 'John Smith' literal value as an argument for name parameter
greet('John Smith')
```

Hello John Smith

```
[9]: greet('Jane')
```

Hello Jane

```
[10]: # argument can be variables as well
n = 'Michael Smith'
greet(n)
```

Hello Michael Smith

```
[11]: n1 = input('Enter your name: ')
greet(n1)
```

Enter your name: John Doe
Hello John Doe

```
[12]: greet('asfdasf') # How to fix? provide either default value or call it properly
```

Hello asfdasf

```
[13]: # function takes one optional argument
def greet(name="Anonymous"):
    print(f'Hello, {name}')
```

```
[14]: # calling greet without an argument
# default value for name will be used!
greet()
```

Hello, Anonymous

```
[15]: greet('adfasd')
```

Hello, adfasd

```
[16]: user = input('Enter your name: ')
greet(user) # calling greet with an argument
```

Enter your name: Jane Smith
Hello, Jane Smith

1.8 3.3.5 Scope of variables

- variable scope tells Python where the variables are visible and can be used

- not all the variables can be used everywhere after they're declared
- Python provides two types of variables or scopes: global and local scopes

1.8.1 global scope

- global variables
- any variables/identifiers defined outside functions
- can be readily accessed/used from within the functions
- must use **global** keyword to update the global variables

1.8.2 local scope

- local variables
- the variables defined in a function have local scope
- can be used/accessed only from within a function after it has been declared
- parameter is also a local variable to the function

1.8.3 global and local scopes demo

Visualize it with [PythonTutor.com](https://pythontutor.com)

```
[17]: # Global and local scope demo
name = "Alice" # global variable
def someFunc(a, b):
    print('name = ', name) # access global variable, name
    name1 = "John" # declare local variable
    print('a = {} and b = {}'.format(a, b)) # a and b are local variables
    print('Hello {}'.format(name1)) # access local variable, name1

someFunc(1, 'Apple')
print(name) # access global variable name
print(name1) # Can you access name1 which is local to someFunc function
```

```
name = Alice
a = 1 and b = Apple
Hello John
Alice
```

```

      □
↩-----
NameError                                Traceback (most recent call
↩last)

<ipython-input-17-2046a00ee4da> in <module>
      9 someFunc(1, 'Apple')
     10 print(name) # access global variable name
--> 11 print(name1) # Can you access name1 which is local to someFunc
↩function
```

```
NameError: name 'name1' is not defined
```

1.8.4 modify global variables from within a function

```
[18]: # how to modify global variable inside function
var1 = "Alice" #global
def myFunc(arg1, arg2):
    global var1 # tell myFunc that var1 is global
    var1 = "Bob" # global or local? How can we access global var1?
    var2 = "John"
    print('var1 = {}'.format(var1))
    print('var2 = ', var2)
    print('arg1 = ', arg1)
    print('arg2 = ', arg2)

myFunc(1, 'Apple')
print(var1)
```

```
var1 = Bob
var2 = John
arg1 = 1
arg2 = Apple
Bob
```

1.8.5 Visualize in PythonTutor.com

1.8.6 Exercise

- Define a function that takes two numbers as arguments and returns the sum of the two numbers as answer

```
[19]: def add(num1, num2):
    """
    takes two numeric values: num1 and num2
    calculates and returns the sum of num1 and num2
    """
    total = num1 + num2
    return total
```

```
[20]: # displays the function prototype and docstring below it
help(add)
```

```
Help on function add in module __main__:
```

```
add(num1, num2)
    takes two numeric values: num1 and num2
```

calculates and returns the sum of num1 and num2

```
[21]: import math
      help(math.sin)
```

Help on built-in function sin in module math:

```
sin(x, /)
    Return the sine of x (measured in radians).
```

```
[22]: # call to test add function
      print(add(100, 200))
```

300

```
[23]: t = add(100.99, -10)
      print('sum = ', t)
```

sum = 90.99

```
[24]: num1 = 15
      num2 = 10.5
      total = add(num1, num2)
      print('{}+{}={}'.format(num1, num2, total))
```

15+10.5=25.5

1.8.7 Exercise

- Define a function that takes two numbers and returns the product of the two numbers.

```
[25]: # Exercise - complete the following function
      def multiply(x, y):
          """
          Function takes two numbers, x and y.
          Returns the product of x and y.
          FIXME
          """
          pass
```

```
[26]: # help can be run for user-defined functions as well
      help(multiply)
```

Help on function multiply in module __main__:

```
multiply(x, y)
    Function takes two numbers, x and y.
```

```
Returns the product of x and y.  
FIXME
```

```
[27]: # Manually test multiply function
```

1.9 3.3.6 Automatic testing of functions / unit testing

- functions can be testing automatically as well as manually
- assert statement can be used to automatically test fruitful functions
- each assertion must be True or must pass in order to continue to the next
- if assertion fails, throws AssertionError exception and program halts

```
[28]: # examples of assert statments  
# == comparison operator that lets you compare two values  
# more on comparison operators in later chapter  
assert True == True
```

```
[29]: assert 10 != '10'
```

```
[30]: assert True == False  
print('this will not be printed')
```

```
↳ -----  
  
      AssertionError                                Traceback (most recent call↳  
↳last)  
  
    <ipython-input-30-7ad819d42cdb> in <module>  
----> 1 assert True == False  
      2 print('this will not be printed')  
  
      AssertionError:
```

```
[31]: assert 'a' == 'A'
```

```
↳ -----  
  
      AssertionError                                Traceback (most recent call↳  
↳last)  
  
    <ipython-input-31-edelacf3c16e> in <module>
```



```
----> 1 assert 'a' == 'A'
```

AssertionError:

```
[32]: # auto testing add function
assert add(2, 3) == 5
assert add(10, -5) == 5
# assert add(100, 2000.99) == ?
```

```
[33]: # unit test multiply function
# write some sample test cases for multiply function using assert statement
```

1.10 3.3.7 Ways of passing data to functions

- data/values are passed to functions in two ways

1.10.1 pass by value

- fundamental types and values (string, int, float) are passed by value by copying the values to the corresponding parameters

1.10.2 pass by reference

- advanced container types (tuple, list, dict, etc.) are passed by reference
- this topic will be discussed in the corresponding chapter covering those container types

```
[ ]: # pass by value demo
var1 = 'John' # global variable
def greetSomeone(para1):
    print('hello', para1)
    var1 = 'Jake' # local variable
    print('hello again', para1)

greetSomeone(var1)
print('var1 = ', var1)
```

1.10.3 visualize pass by value with PythonTutor.com

1.11 3.3.8 Fruitful functions returning multiple values

- functions can return more than 1 values
- multiple comma separated values can be returned
- the values are return as Tuple type (more on this later)

```
[34]: def findAreaAndPerimeter(length, width):
      """
      Function takes length and width of a rectangle.
```

```

Finds and returns area and perimeter of the rectangle.
"""
area = length*width
perimeter = 2*(length+width)
return area, perimeter

```

```
[35]: print(findAreaAndPerimeter(10, 5))
```

```
(50, 30)
```

```
[36]: a, p = findAreaAndPerimeter(20, 10)
print(f'area = {a} and perimeter = {p}')
```

```
area = 200 and perimeter = 60
```

```
[37]: # test getAreaAndParameter() function
assert findAreaAndPerimeter(4, 2) == (8, 12)
```

1.12 3.3.9 Function calling a function

- a function can be called from within another function
- a function can call itself – called recursion (see Chapter 13)

```
[38]: def average(num1, num2):
      sum_of_nums = add(num1, num2)
      return sum_of_nums/2
```

```
[39]: avg = average(10, 20)
print(f'avg of 10 and 20 = {avg}')
```

```
avg of 10 and 20 = 15.0
```

1.13 3.3.10 Exercises

1.13.1 exercise 1

Write a function that takes two numbers; subtracts the second from the first and returns the difference. Write two test cases.

```
[40]: # solution to exercise 1
def sub(num1, num2):
    return num1-num2
```

```
[41]: def test_sub():
      assert sub(100, 50) == 50
      assert sub(80, 45.5) == 34.5
      print('all test cases passed for sub()!')
```

```
[42]: test_sub()
```

all test cases passed for sub()!

1.13.2 exercise 2

```
[43]: def get_time(seconds):  
      pass
```

```
[44]: #Here are some tests that should pass:  
def test_get_time():  
    assert get_time(3600) == '1:0:0'  
    assert get_time(3661) == '1:1:1'  
    assert get_time(3666) == '1:1:6'  
    assert get_time(36610) == '10:10:10'  
    print('all test cases passed for get_time()')
```

```
[45]: test_get_time()
```

```
      ␣  
↪-----  
  
      AssertionError                                Traceback (most recent call␣  
↪last)  
  
      <ipython-input-45-dae45a4e0ae1> in <module>  
      ----> 1 test_get_time()  
  
      <ipython-input-44-0c9bcf598e1c> in test_get_time()  
          1 #Here are some tests that should pass:  
          2 def test_get_time():  
      ----> 3     assert get_time(3600) == '1:0:0'  
          4     assert get_time(3661) == '1:1:1'  
          5     assert get_time(3666) == '1:1:6'  
  
      AssertionError:
```

1.13.3 exercise 3

```
[46]: def hypotenuse(leg1, leg2):  
      pass
```

```
[47]: def test_hypotenuse():  
      assert hypotenuse(3, 4) == 5.0  
      assert hypotenuse(12, 5) == 13.0  
      assert hypotenuse(24, 7) == 25.0
```

```
assert hypotenuse(9, 12) == 15.0
print('all test cases passed hypotenuse()')
```

```
[48]: test_hypotenuse()
```

```

      □
↳-----

AssertionError                                Traceback (most recent call↳
↳last)

<ipython-input-48-bbecdabdf81> in <module>
----> 1 test_hypotenuse()

<ipython-input-47-d9a390da2f14> in test_hypotenuse()
      1 def test_hypotenuse():
----> 2     assert hypotenuse(3, 4) == 5.0
      3     assert hypotenuse(12, 5) == 13.0
      4     assert hypotenuse(24, 7) == 25.0
      5     assert hypotenuse(9, 12) == 15.0
```

AssertionError:

1.13.4 exercise 4

```
[49]: def slope(x1, y1, x2, y2):
      pass
```

```
[50]: def test_slope():
      assert slope(5, 3, 4, 2) == 1.0
      assert slope(1, 2, 3, 2) == 0.0
      assert slope(1, 2, 3, 3) == 0.5
      assert slope(2, 4, 1, 2) == 2.0
      print('all test cases passed for slope()')
```

```
[51]: test_slope()
```

```

      □
↳-----

AssertionError                                Traceback (most recent call↳
↳last)
```

```
<ipython-input-51-329d174a7258> in <module>
----> 1 test_slope()
```

```
<ipython-input-50-b2c25a2ea8d1> in test_slope()
1 def test_slope():
----> 2     assert slope(5, 3, 4, 2) == 1.0
3     assert slope(1, 2, 3, 2) == 0.0
4     assert slope(1, 2, 3, 3) == 0.5
5     assert slope(2, 4, 1, 2) == 2.0
```

AssertionError:

```
[52]: def intercept(x1, y1, x2, y2):
      pass
```

```
[53]: def test_intercept():
      assert intercept(1, 6, 3, 12) == 3.0
      assert intercept(6, 1, 1, 6) == 7.0
      assert intercept(4, 6, 12, 8) == 5.0
      print('all test cases passed for intercept()')
```

```
[54]: test_intercept()
```

```

      □
↪-----
AssertionError                                Traceback (most recent call
↪last)
```

```
<ipython-input-54-4c3f95bc59e3> in <module>
----> 1 test_intercept()
```

```
<ipython-input-53-4b8e8188161f> in test_intercept()
1 def test_intercept():
----> 2     assert intercept(1, 6, 3, 12) == 3.0
3     assert intercept(6, 1, 1, 6) == 7.0
4     assert intercept(4, 6, 12, 8) == 5.0
5     print('all test cases passed for intercept()')
```

AssertionError:

1.14 Kattis problems requiring functions

- functions are not required to solve problems
- you can use function to solve each and every problem
- function is required if you must write automated unit tests

[]: