python™

FUNCTIONS

Prof. K. Adisesha

BE, M.Sc., M.Th., NET, (Ph.D.)

Learning objectives

- Understanding Functions
- Defining Functions in Python
- Flow of Execution in a Function Call
- Passing Parameters
- Returning Values from Functions
- Composition
- Scope of Variables

What Are Functions?

- A function is a block of code which only runs when it is called.
- Functions are sub-programs which perform tasks which may need to be repeated.
- Some functions are "bundled" in standard libraries which are part of any language's core package. We've already used many built-in functions, such as input(), eval(), etc.
- Functions are similar to methods, but may not be connected with objects
- Programmers can write their own functions

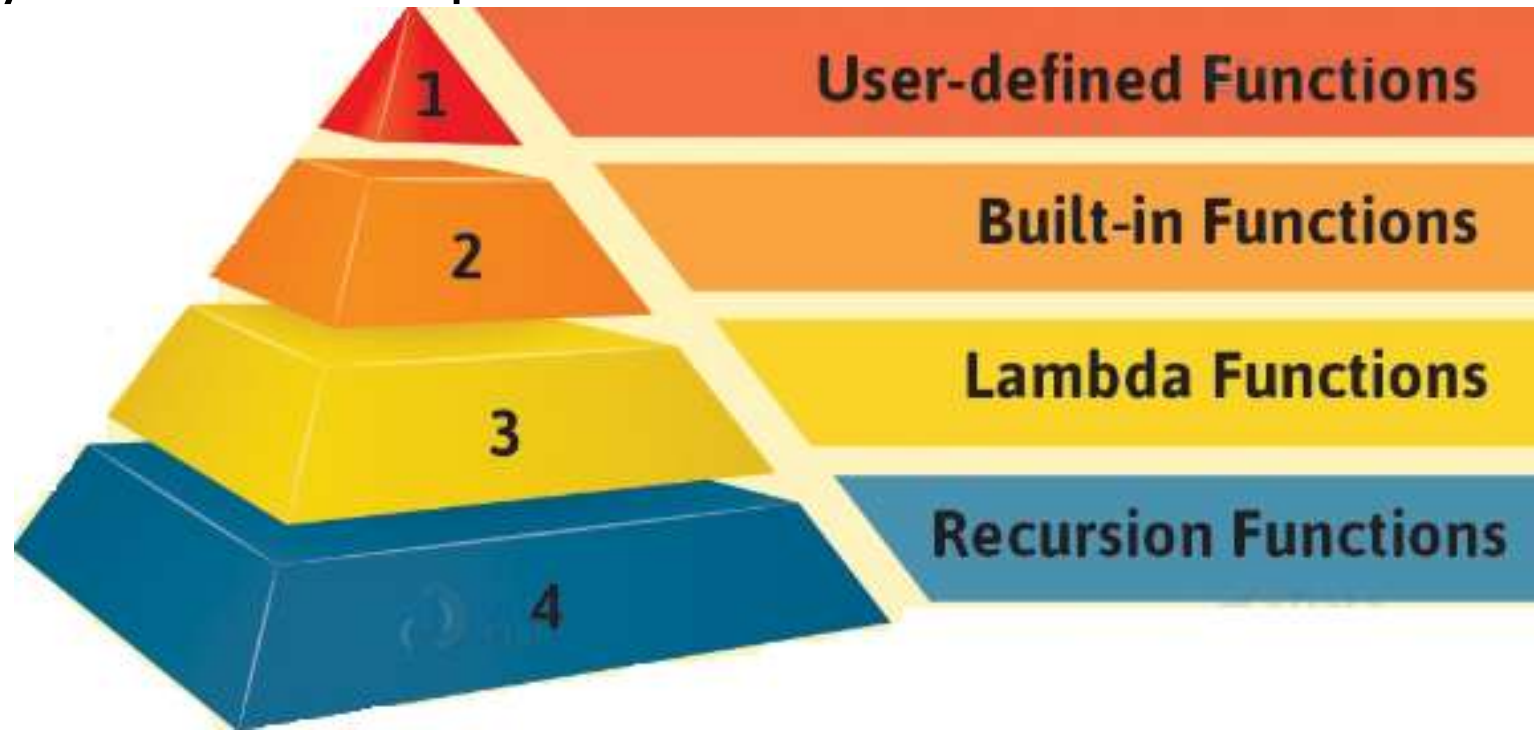
Why Write Functions?

- Reusability
- Fewer errors introduced when code isn't rewritten
- Reduces complexity of code
- Programs are easier to maintain
- Programs are easier to understand

Types of Functions

Different types of functions in Python:

Python built-in functions, Python recursion function, Python lambda function, and Python user-defined functions with their syntax and examples.



User-Defined Functions

- Python lets us group a sequence of statements into a single entity, called a function.
- A Python function may or may not have a name.

Advantages of User-defined Functions in Python

- This Python Function help divide a program into modules. This makes the code easier to manage, debug, and scale.
- It implements code reuse. Every time you need to execute a sequence of statements, all you need to do is to call the function.
- This Python Function allow us to change functionality easily, and different programmers can work on different functions.

Function Elements

- Before we can use functions we have to define them.
- So there are two main elements to functions:
  1. Define the function. The function definition can appear at the beginning or end of the program file.

```
def my_function():
    print("Hello from a function")
```

  2. Invoke or call the function. This usually happens in the body of the main() function, but sub-functions can call other sub-functions too.

```
main():
    my_function()
```

Rules for naming function (identifier)

- We follow the same rules when naming a function as we do when naming a variable.
- It can begin with either of the following: A-Z, a-z, and underscore(_).
- The rest of it can contain either of the following: A-Z, a-z, digits(0-9), and underscore(_).
- A reserved keyword may not be chosen as an identifier.
- It is good practice to name a Python function according to what it does.

Function definitions

- A function definition has two major parts: the definition head and the definition body.
- The definition head in Python has three main parts: the keyword def, the identifier or name of the function, and the parameters in parentheses.

```
def average(total, num):
```

- def - keyword
- Average-- identifier
- total, num-- Formal parameters or arguments
- Don’t forget the colon : to mark the start of a statement bloc

Function body

- The colon at the end of the definition head marks the start of the body, the bloc of statements. There is no symbol to mark the end of the bloc, but remember that indentation in Python controls statement blocs.

```
def average(total, num):
    x = total/num          #Function body
    return x               #The value that's returned when the
                             function is invoked
```

Workshop

Using the small function defined in the last slide, write a command line program which asks the user for a test score total and the number of students taking the test. The program should print the test score average.

- Example: Function Flow - happy.py

```
# Simple illustration of functions.
def happy():
        print "Happy Birthday to you!"
def sing(person):
        happy()
        print "Happy birthday, dear", person + "."
def main():
        sing("Sunny")
        print
main()
```

Parameters or Arguments

- The terms parameter and argument can be used for the same thing: information that are passed into a function.
- From a function's perspective:
  - A parameter is the variable listed inside the parentheses in the function definition.
  - An argument is the value that are sent to the function when it is called.
- Arguments are often shortened to args in Python documentations.
- By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

```
def my_function(fname, lname):   <-------- Parameters
  print(fname + " " + lname)

my_function("Narayana", "College")   <-------- Arguments
```

Arbitrary Arguments, *args

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.
- This way the function will receive a *tuple* of arguments, and can access the items accordingly:

Example: If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*name):
  print("The youngest child is " + name[2])

my_function("Rekha", "Prajwal", "Sunny")
```

Output: The youngest child is Sunny

- If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.
- This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

```
def my_function(**name):
  print("His last name is " + name["lname"])

my_function(fname = "Prajwal", lname = "Sunny")
```

Output: His last name is Sunny

Formal vs. Actual Parameters (and Arguments)

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

```
# moveto.py
from graphics import *
def moveTo(object, point):
 c = object.getCenter()
dx = point.getX() - c.getX()
dy = point.getY() - c.getY()
object.move(dx,dy)

def main():
win = GraphWin()
circ = Circle(Point(100,100), 20)
circ.draw(win)
p = win.getMouse()
moveTo(circ, p)
win.close()
center = circ.getCenter()
print center.getX(), center.getY()
main()
```

Formal Parameters

Function definition

Actual parameters or arguments

Call or invocation of function; Arguments must be in correct order according to function definition

Scope of variables

- A variable's scope tells us where in the program it is visible. A variable may have local or global scope.
- Local Scope- A variable that's declared inside a function has a local scope. In other words, it is local to that function.
- Thus it is possible to have two variables named the same within one source code file, but they will be different variables if they're in different functions—and they could be different data types as well.

```
>>> def func3():
        x=7
        print(x)
>>> func3()
```

- Global Scope- When you declare a variable outside python function, or anything else, it has global scope. It means that it is visible everywhere within the program.

```
>>> y=7
>>> def func4():
        print(y)
>>> func4()
```

Scope of variables, cont.

```
def calc_tax(x):
    x = x * 0.08
    return x

def add_shipping(subtot):
    subtot = subtot * 1.04
    return subtot

def main():
    units = input("Please enter the # of units")
    firstTotal = units * 5.00
    total = add_shipping(firstTotal)
    total = total + calc_tax(total)
    print "Your total is: ", total
 main()
```

x has scope only in calc_tax function

subtot has local scope only

Invocation/call

firstTotal is sent as a parameter, and returns a value stored in total

Functions: Return values

- Some functions don't have any parameters or any return values, such as functions that just display. But…
- "return" keyword lets a function return a value, use the return statement:

```
def square(x):              # x is Formal parameter
    return x * x            #Return value
```

- The call: output = square(3)

The pass Statement

- function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

Example

```
def myfunction():
  pass
```

Return value used as argument:

- Example of calculating a hypotenuse

```
num1, num2 = 10, 14
Hypotenuse = math.sqrt(sum_of_squares(num1, num2))
def sum_of_squares(x,y):
        t = (x*x) + (y * y)
        return t
```

Returning more than one value

- Functions can return more than one value

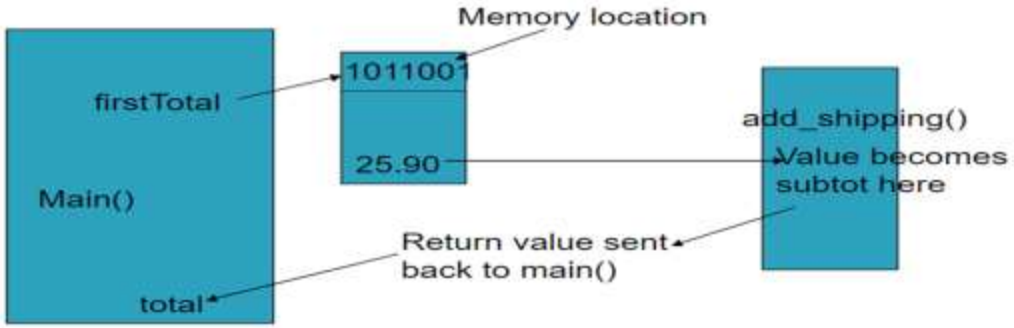
```
def hi_low(x,y):
        if x >= y:
            return x, y
        else: return y, x
```

- The call:

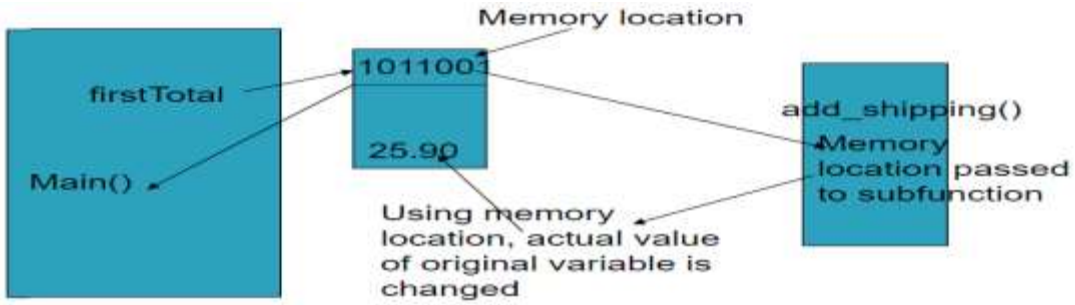
```
        hiNum, lowNum = hi_low(data1, data2)
```

Modifying parameters, cont.

- Some programming languages, like C++, allow passing parameters by reference. Essentially this means that special syntax is used when defining and calling functions so that the function parameters refer to the memory location of the original variable, not just the value stored there.
- Schematic of passing by value



- PYTHON DOES NOT SUPPORT PASSING PARAMETERS BY REFERENCE



Schematic of passing by reference

Passing lists in Python

- Python does NOT support passing by reference, BUT…
- Python DOES support passing lists, the values of which can be changed by subfunctions.
- Example of Python's mutable parameters

```
#    Illustrates modification of a mutable parameter (a list).

def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)

def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, 0.05)
    print amounts

test()
```

Passing lists, cont.

- Because a list is actually a Python object with values associated with it, when a list is passed as a parameter to a subfunction the memory location of that list object is actually passed –not all the values of the list.
- When just a variable is passed, only the value is passed, not the memory location of that variable.
- E.g. if you send a List as an argument, it will still be a List when it reaches the function:
- Example

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

Output:

```
apple
banana
cherry
```

Recursion

- Python also accepts function recursion, which means a defined function can call itself.
- Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.
- The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.
- Example:

```
def Recursion(k):
  if(k > 0):
    result = k + Recursion(k - 1)
    print(result)
  else:
    result = 0
  return result
print("\n\nRecursion Example Results")
Recursion(6)
```

Python Lambda

- A lambda function is a small anonymous function.
- The power of lambda is better shown when you use them as an anonymous function inside another function.
- A lambda function can take any number of arguments, but can only have one expression.
- Syntax

  ```
  lambda arguments : expression
  ```

- The expression is executed and the result is returned:
- Example
  - A lambda function that adds 10 to the number passed in as an argument, and print the result:

    ```
    x = lambda a : a + 10
    print(x(5))
    ```

Output: 15

Conclusion!

- We learned about the Python function.
- Types of Functions
- Advantages of a user-defined function in Python
- Function Parameters, arguments and return a value.
- We also looked at the scope and lifetime of a variable.

Thank you