#8: Functions

Reading: Chapter 6
Contents

Function Review
• Definition, implementation, function call
• Input parameters - keyword arguments, default values
• Value returning functions
• Scope of variables
• Local and global variables
• Global constants

Functions
• Pass by reference versus pass by value

NOTE: From now on, we will use only python3 for the class!
Functions

So far, your Python program was one continuous list of instructions

```python
a = 10  # 1. assign values to variables
b = 5

c = a + b  # 2. calculate sum of variables
print(c)  # 3. print resulted sum

b = 0.5  # 4. repeat steps 1-3

c = a + b
print(c)
```

You can simplify this by dividing the whole work into several smaller tasks, which each becomes a function:

```python
def sumNum(a,b):
    c = a + b
    print(c)
    return c

# main code – function calls
sumNum(10,5)
sumNum(10,0.5)
```
Functions

Purpose:

• Code reusability
• Your programs become much more readable
• Better code organization
• Less errors – if similar functionalities more than once in your program just create and invoke a function many times (better to test, less bugs, shorter code, etc)

**Functions are programmer created code blocks that do a specific task.**

Function are same as *black boxes* - Once a function is defined, one can use it without knowing how it was implemented (the code inside). Functions encapsulate details.
Function Definition

**Function** - performs a specific task

Syntax:

```python
def functionname([[parameters]]):
    #function_docstring #documentation
    function_Statement(s)
    return [expression]
```
Function call

# Function definition is here

def printme(strg):
    #This prints a passed string into this function
    print("Input param: ", strg)
    return

# in main – put function call
printme("I'm first call to user defined function!")
printme("Again second call to function")

Input param: I'm first call to user defined function!
Input param: Again second call to function
Input parameters with positional arguments

By default, input parameters have a **positional behavior**, and you need to inform them in the **same order and number** that they were defined.

```python
def birthday1(name, age):
    print("Happy BDay ",name,"! You’re ",age," today!")
birthday1("Lisa",5)
Happy BDay Lisa! You’re 5 today!
birthday1(1,"Lisa") # Position of each parameter is important!
Happy BDay 1! You’re Lisa today!
birthday1() #error – must have the same number of parameters as in the function definition
```

For correct function call make sure each parameter has its correct position and there are the same number of parameters in both function call and function definition!
def dividesafe(a, b):
    # Handle zero denominator
    if b == 0:
        return -1
    # Divide
    return a / b

# Use method
print(dividesafe(10, 5))
print(dividesafe(10, 0))
2.0
-1
Input parameters with keyword arguments

When you use **keyword arguments in a function call**, the caller identifies the arguments by the parameter name.

Features:

- You can place **arguments out of order**
- You are **not allowed to skip arguments**

```python
def birthday2(name, age):
    print(“Happy BDay “,name,”! You’re “,age,” today!”)

birthday2(name=“Lisa”, age=5)
Happy BDay Lisa! You’re 5 today!
birthday2(age=10)  # error – must have exactly 2 arguments
birthday2(age=2, name=“Joe”) #order doesn’t matter
Happy BDay Joe! You’re 2 today!
```
Mixing keywords and positionals

You can combine keyword with positional arguments in the same function call, but once you used a keyword argument, all the following arguments must be keyword arguments.

```python
def display_message(message, start, end):
    print message[start:end]

# main code
display_message("message", start = 0, end = 3)
mes
# following is not syntactically correct
display_message(end=5, start=1, "my message")
```
Input parameters with default values

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

No need for same order or same number of arguments.

def birthday3(name="Lisa", age=5):
    print("Happy BDay ",name,"! You’re ",age," today!")
birthday3()
Happy BDay Lisa! You’re 5 today!
birthday3(age=10)
Happy BDay Lisa! You’re 10 today!
birthday3(age=2,name="Joe")  #order doesn’t matter
Happy BDay Joe! You’re 2 today!

Suggestion: use one (either) type of parameters till you get used to them well.
No value returning function

Methods in Python do not have to return a value. If we simply use the "return" statement alone, no value is returned.

```python
def printname(first, middle, last):
    # Validate middle initial length.
    if len(middle) != 1:
        print("Middle initial too long")
        return

    # Display.
    print(first + " " + middle + ". " + last)

# Call method.
printname("Jake", "R", "Chambers")
Jake R. Chambers
```
Value returning function

The statement return [expression] exits a function, optionally passing back an expression to the caller.

```python
def square(x):
    #define the function
    return x * x  #the square of a number

# main
for y in range(1, 5):  #cycle through a list of numbers
    print(square(y))  #print the square of a number

1
4
9
16
```
Program structure

#Function definitions
def fct1(...):
    body_fct1
def fct2(...):
    body_fct2

#Main code which includes function calls
statement1  # usually, input from user, values assign to variables
fct1(...)  #call of fct1
fct2(...)  #call of fct2
statement2
statement3
....
Scope of variable

• The scope of a variable determines the portion of the program where you can access it.

• Two basic scopes of variables in Python:
  – Global variables
  – Local variables

• Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

• This means that local variables can be accessed only inside the function in which they are declared whereas global variables can be accessed throughout the program body by all functions.
Local variables

def m():
    x = 3      #local variable
    f(x)

def f(x):
    print(x)  # no error: x here is not the same
               # as x in m()

# main
m()
3

Suggestion: To avoid confusion, use different names for each of the variables, this way you can easily see their scope.
Global variables

To modify a global variable in the function scope we need a new keyword `global`

```python
a = "global"
def f():
    global a # starting this point you can modify the value of the global variable inside f
    a = "modified in local scope"
    print("in f() value of a is:", a)
#main
print("value of a is:", a)
f()
print("value of a is:", a)
value of a is: global
in f() value of a is: modified in local scope
value of a is: modified in local scope
```
Global constants

It is good programming practice to avoid defining global variables and instead to put your variables inside functions and explicitly pass them as parameters where needed.

One common exception is constants. It is a good practice to have global constants and use them throughout the entire program.

By convention, names for constants are all capital letters.
Pass by object reference

- **Passing** input parameters (arguments) **by object reference** means that if you change what a parameter refers to within a function, **the change** also **reflects back** in the calling function.

**Mutable objects** (like lists) can be passed by reference.

```python
def changemeRef(mylist):
    mylist.append([1,2,3,4])
    print("Values inside the function: ", mylist)    # return

# Now you can call changemeRef function
mylist = [10,20,30]
changemeRef( mylist )
print("Values outside the function: ", mylist)
```

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
Pass by value

Parameter **passed by value** means that if you change what a parameter refers to within a function, the change does not reflect back in the calling function.

```python
def changemeVal(mylist):
    mylist = [1,2,3,4]  # This would assign new reference in mylist
    print("Values inside the function: ", mylist)  # return

# Now you can call changeme function
mylist = [10,20,30]
changemeVal(mylist)
print("Values outside the function: ", mylist)
```

Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]
Next lecture ... 

- Functions
- Pass by reference versus pass by value
- Anonymous functions (lambda)
- Variable-length arguments
- Recursive functions