#7: Functions

Reading: Chapter 6
Contents

Functions

• Definition, implementation, function call
• Input parameters
• Value returning functions
• Scope of variables
• Local and global variables
• Global constants

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

Python programs are comprised of functions, classes, modules, and packages.

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

- **Classes** are object-oriented structures - pretty powerful structures that can make programming life easier.

- **Modules** are generally considered normal program files, i.e. a file comprised of functions/classes, loops, control statements, etc.

- **Packages** are programs made up of many different modules.
Functions

So far, your Python programs were one, continues list of instructions: variables and data structures, use of variables, conditions, loops, function calls

Developing programs is a hard job. You can simplify this by dividing the whole work into several smaller tasks. The tasks become functions

Purpose:
• Better code organization
• Code reusability
• Your programs become much more readable
• 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.

Examples:
print(“String 1”, “ String 2”) #output is: String 1 String 2
range(5)
Definition

**Function** is an encapsulated small program that has input (optional), does something (usable), and returns something (optional) – exactly like a program.

**Syntax:**

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

Rules:

• Function blocks begin with the keyword `def` followed by the function name and parentheses ( ).
• Any input parameters or arguments should be placed within these parentheses.
• The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
• The code block within every function starts with a colon “:” and is indented.
• The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`. The expression can have none, one or more parameters separated by comma “,”.
Definition

```python
# Function definition is here
def printme(str):
    #This prints a passed string into this function
    print("Input parameter ", str)
    return
```

Defining a function only gives it a name, specifies the parameters that are to be included in the function, uses these parameters to perform a task and returns something.

NOTE: One must first define a function before using it!
Function call

# Function definition is here
def printme(str):
    #This prints a passed string into this function
    print(“Input param: “, str)
    return
# Now you can call printme function
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

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!
Input parameters example

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. This allows you to place arguments out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

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!
```
**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.

```python
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.
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.

def square(x):
    return x * x

for y in range(1, 5):
    print(square(y))
def sum(arg1, arg2):
    # Add both the parameters and return them.
    total = arg1 + arg2
    print("Inside the function : ", total)
    return total

# Now you can call sum function
total = sum(10, 20)
print("Outside the function : ", total)

Inside the function : 30
Outside the function : 30
def fullname(str):
    # Returns first, middle and last names.
    aux = str.split()
    first = aux[0]
    middle = aux[1]
    last = aux[2]
    return first, middle, last

# Now you can call sum function
first1, middle1, last1 = fullname("Joe R. Smith")
print("Outside the function last name: ", last1)
Outside the function last name: Smith
Program structure

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

#Main code which includes function calls
statement1
fct1(...) #call of fct1
fct2(...) #call of fct2
statement2
statement3
....
Scope of variable

• All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

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

• There are two basic scopes of variables in Python:
  – Global variables
  – Local variables
Local vs. Global 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 main():
    x = 3  # local variable
    f()

def f():
    print(x)  # error: f does not know about the x defined in main

#...main code
main()
Local variables

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

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

main()

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)

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.
Global constants

PI = 3.14159265358979  # global constant

def circleArea(radius):
    return PI * radius * radius

def circleCircumference(radius):
    return 2 * PI * radius

print('circle area with radius 5:', circleArea(5))
print('circumference with radius 5:', circleCircumference(5))

circle area with radius 5: 78.539816339
circumference with radius 5: 31.4159265358
Next lecture ...

- continue on functions