Chapter 6: Functions

- A **function** is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result. A function is a group of statements performing a specific task.
- When a program gets bigger in size and its complexity grows, it gets difficult for a programmer to keep track of which piece of code is doing what!
- A **function** can be reused by the programmer in a given program any number of times.

Creating a Function

In Python a function is defined using the `def` keyword:

Example:

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

Calling a Function

To call a function, use the function name followed by parenthesis:

```
my_function()
```

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.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):
  print("Name is ",fname )

my_function("Ravi")
my_function("Raju")
my_function("Kamal")
```

Parameters / 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 is sent to the function when it is called.

In the above example, fname is the **parameter** and "Ravi" is the **argument**

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function has 2 parameters, you have to call the function with 2 arguments, not more, and not less. Otherwise you will receive an error

The following function has 2 parameters and hence is called with 2 arguments

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

my_function("Ravi", "Nallan")
my_function("Ravi") # Error
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
```

```
my_function()
my_function("Brazil")
```

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:

```
def add(*param):
        print("number of parameters is ",len(param))
add(1,2)
add()
add(1)
```

Another Example: Add numbers based on no. of params

```
def add(*i):
    if len(i)==0:
        print("nothing to add " )
    elif len(i)==1:
        print("Addition of 1 element")
    elif len(i)==2:
        print("Addition of 2 element",i[0]+i[1])
add(1,2)
add()
add(1)
```

Keyword Arguments

You can also send arguments with the *key = value* syntax.

This way the order of the arguments does not matter.

```
def my_function(param1, param2, param3):
  print("The param3 is " + param3)

my_function(param1= "Parameter1", param2= " Parameter2", param3= " Parameter3")
```

Arbitrary Keyword Arguments, **kwargs

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(**kwargs):
  print("His last name is " + kwargs ["lname"])

my_function(fname = "Ravi", lname = "Nallan")
```

Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

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

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Return Values

To let a function return a value, use the `return` statement:

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Functions Need not return a value

```
def my_function(x):
  x = 5 * x
  print(x)

print(my_function(3))    # prints None
```

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.

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

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(infinite loop), 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.

In this example, `factorial()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements (`-1`) every time we recurse. The recursion ends when the condition is equal to 1.

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

```
def factorial(k):

    if x == 1:
        return 1
    else:
        return (k * factorial(k-1))

num = 3
print("The factorial of", num, "is", factorial(num))
```

Types of Functions

There are two types of functions in Python:

- **Built-in** functions #Already present in Python
- **User-defined** functions #Defined by the user

Examples of built-in function includes len(), print(), range(), etc.

The functions we defined above are examples of a user-defined function.

Exercises

Problem 1: Write a Python function to sum 2 numbers

Problem 2: Write a Python function to multiply all the numbers in a list.

Problem 3: Write a Python function to calculate the factorial of a number (a non-negative integer). The function accepts the number as an argument.
Problem 4: Write a Python program to get the largest number from a list.(Use sort())

Problem 5: Python program to print the numbers from a given number n till 0 using recursion

Problem 6: Python program to check whether a string is palindrome or not

Exercise 7: Python program to implement a calculator to do basic operations. Accept the numbers and the operation(+,-,*) from the input. Write 1 method for each operation

Exercise 8: Write a Python function(which accepts 2 parameters, tag and content) to create the HTML string with tags around the word(s).
Input : Tag = body. content=Python

Tag = i. content=Italic tag

Output:

```
<body>
        Python
</body>
<i>
        Italic tag
</i>
```

Exercise 9: Write a Python program to get a list, sorted in increasing order by the last element in each tuple from a given list of non-empty tuples.

Exercise 10: Write a Python program to print the even numbers from a given list.

Exercise 11: Write a Python program that accepts a hyphen-separated sequence of words as input and prints the words in a hyphen-separated sequence after sorting them alphabetically.

Sample Items : green-red-yellow-black-white
Expected Result : black-green-red-white-yellow

Exercise 12: Write a Python function to create and print a list where the values are square of numbers between 1 and 30 (both included).

Variable Scope in Python

A variable is only available from inside the region it is created. This is called **scope**.

Local Scope

A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

```
def myfunc():
  x = 300       # Local scope , Available inside the function only
  print(x)

myfunc()
```

Function Inside Function

As explained in the example above, the variable `x` is not available outside the function, but it is available for any function inside the function:

```
def myfunc():
  x = 300
  def myinnerfunc():
    print(x)
  myinnerfunc()

myfunc()
```

Global Scope

A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

```
x = 300

def myfunc():
  print(x)

myfunc()

print(x)
```

Naming Variables

If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):

```
x = 300

def myfunc():
  x = 200
  print(x)

myfunc()

print(x)
```

Global Keyword

If you need to create a global variable, but are stuck in the local scope, you can use the `global` keyword.

The `global` keyword makes the variable global.

```
def myfunc():
  global x
  x = 300

myfunc()

print(x)
```

Also, use the `global` keyword if you want to make a change to a global variable inside a function.
To change the value of a global variable inside a function, refer to the variable by using the global keyword:

```
x = 300

def myfunc():
  global x
  x = 200

myfunc()

print(x)
```

The “nonlocal” statement

Without “nonlocal” statement

```
def foo():
    x = "enclosed"
    def bar():
        x = "local"
    bar()
    print(x)  # gives "enclosed"
foo()
```

You can assign a value to an enclosed variable with the nonlocal statement:

```
def foo():
    x = "enclosed"
    def bar():
        nonlocal x
        x = "local"
    bar()
    print(x)  # gives "local"
foo()
```

Some Examples:

```
xs = []
def foo():
    xs.append(42)   # OK foo()
-----------------------------
xs = []
def foo():
    xs =xs+[42]  # UnboundLocalError: local variable 'xs'
                 # referenced before assignment foo()
```

Python Lambdas

- A lambda function is a small anonymous function.

- A lambda function can take any number of arguments, but can only have one expression.
- There is no return statement which is usually present in the def function syntax.
- The function will simply return the expression value even when there is no return statement.

Need for Lambda Functions

There are at least 3 reasons:

- Lambda functions reduce the number of lines of code when compared to normal python function defined using def keyword. But this is not exactly true because, even functions defined with def can be defined in one single line. But generally, def functions are written in more than 1 line.

- They are generally used when a function is needed temporarily for a short period of time, often to be used inside another function such as filter, map and reduce.

- Using lambda function, you can define a function and call it immediately at the end of definition. This can't be done with def functions.

Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned:

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

is equivalent to
```

```
def x(a):
    return a+10
```

Do the same in a single line.

```
# calculate squares using def in one line

def squares_def(x): return x*x

print('Using def: ', squares_def(5))
# calculate squares using lambda in one line

squares = lambda a : a * a
print('Using def: ', squares(5))
```

See that while using the def keyword, we are returning a certain value x*x . In the case of lambda function, the expression x*x will be returned without writing an explicit return statement. Generally in normal use, there is not much of a difference in using def and lambda keyword. Both of them are in fact functions. Let's see their types.

```
# Types

print(type(squares))

print(type(squares_def))

<class 'function'>

<class 'function'>
```

Both of them belong to the class `function`.

You can assign a lambda function to a variable and call the variable x as if it were a function

Lambda functions can take any number of arguments:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Lambda functions can be Immediately Invoked

You can implement a lambda function without using a variable name. You can also directly pass the argument values into the lambda function right after defining it using parenthesis. This cannot be done using def functions.

```
(lambda x,y : x*y)(5,7)

#> 7
```

This doesn't work with `def` function.

```
# Doesn't work with def

def multiply(x, y): return x*y (5,7)
```

Lambda functions accept all kinds of arguments, just like normal def function

`lambda` function supports all kinds of arguments just like the normal `def` function.

Keyword Arguments: `keyword argument` is an argument preceded by an identifier (e.g. name=) in a function call. **Named Arguments: Example**

```
(lambda x, y=3, z=5: x*y*z)(7)

#> 105
```

Variable list of Arguments: Example

```
(lambda x, y=3, z=5: x*y*z)(x=7)

#> 105
```

Variable list of keyword arguments: Example

```
(lambda *args : sum(args))(3,5,7)
```

You can use `lambda` function in `map()`

`map()` function applies a given function to all the items in a list and returns the result. Similar to `filter()`, simply pass the lambda function and the list (or any iterable, like tuple) as arguments.

```
# using lambda inside map function

mylist = [2,3,4,5,6,7,8,9,10]
```

```
list_new  = list(map(lambda x : x%2, mylist))

print(list_new)

#> [0, 1, 0, 1, 0, 1, 0, 1, 0]
```

You can use `lambda` function in `filter()`

`filter()` function is used to filter a given iterable (list like object) using another function that defines the filtering logic. A lambda function is typically used to define the filtering logic and is passed as the first argument of `filter()`. An iterable like a list object is passed as the second argument to the `filter` function.

```
# Using lambda inside filter function

mylist = [2,3,4,5,6,7,8,9,10]

list_new  = list(filter(lambda x : (x%2==0), mylist))

print(list_new)

#> [2, 4, 6, 8, 10]
```

You can use `lambda` function in `reduce()` as well

`reduce()` function performs a repetitive operation over the pairs of the elements in the list. Pass the `lambda` function and the list as arguments to the `reduce()` function. For using the `reduce()` function, you need to import `reduce` from `functools` librray.

```
# Using lambda inside reduce

from functools import reduce

list1 = [1,2,3,4,5,6,7,8,9]

sum = reduce((lambda x,y: x+y), list1)

print(sum)

#> 45
```

See that the `reduce()` function iteratively multiplies over the elements in the list . `i.e 1+2, 1+2+3 , 1+2+3+4` and so on.

Exercises

Problem 1**:** Write a Python program to create a lambda function that adds 15 to a given number passed in as an argument, also create a lambda function that multiplies argument x with argument y and print the result.
Problem 2**:** Write a Python program to create a function that takes one argument, and that argument will be multiplied with an unknown given number.
Sample Output:
Double the number of 15 = 30
Triple the number of 15 = 45
Quadruple the number of 15 = 60
Quintuple the number 15 = 75

Problem 3**:** Write a Python program to sort (on 2nd element) a list of tuples using Lambda.
Original list of tuples:
[('English', 88), ('Science', 90), ('Maths', 97), ('Social sciences', 82)]
Sorting the List of Tuples:
[('Social sciences', 82), ('English', 88), ('Science', 90), ('Maths', 97)]

Problem 4**:** Write a Python program to sort(on color) a list of dictionaries using Lambda.
Original list of dictionaries :
[{'make': 'Nokia', 'model': 216, 'color': 'Black'}, {'make': 'Mi Max', 'model': '2', 'color': 'Gold'}, {'make': 'Samsung', 'model': 7, 'color': 'Blue'}]
Sorting the List of dictionaries :
[{'make': 'Nokia', 'model': 216, 'color': 'Black'}, {'make': 'Samsung', 'model': 7, 'color': 'Blue'}, {'make': 'Mi Max', 'model': '2', 'color': 'Gold'}]

Problem 5**:** Write a Python program to filter a list of integers using Lambda.
Original list of integers:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Even numbers from the said list:
[2, 4, 6, 8, 10]
Odd numbers from the said list:
[1, 3, 5, 7, 9]

Problem 6: Write a Python program to find if a given string starts with a given character using Lambda.
String =Python if it start with P True else False
Sample Output:
True
False
Problem 7: Write a Python program to count the even, odd numbers in a given array of integers using Lambda.
Original arrays:
[1, 2, 3, 5, 7, 8, 9, 10]
Number of even numbers in the above array: 3
Number of odd numbers in the above array: 5

Problem 8: Write a Python program to print the values of list whose length is six using Lambda.
Input : `weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']`

Sample Output:
Monday
Friday
Sunday

Problem 9: Write a Python program to add two given lists using map and lambda.
Original list:
[1, 2, 3]
[4, 5, 6]
Result: after adding two list
[5, 7, 9]
Problem 10: Write a Python program to find palindromes in a given list of strings using Lambda.
Orginal list of strings:
['php', 'w3r', 'Python', 'abcd', 'Java', 'aaa']
List of palindromes:
['php', 'aaa']
Problem 11: Write a Python program to compute sum of digits of a given string.