Functions in Python

In this part of the Python programming tutorial, we will talk about functions.

A function is a piece of code in a program. The function performs a specific task. The advantages of using functions are:

- Reducing duplication of code
- Decomposing complex problems into simpler pieces
- Improving clarity of the code
- Reuse of code
- Information hiding

Functions in Python are first-class citizens. It means that functions have equal status with other objects in Python. Functions can be assigned to variables stored in collections, or passed as arguments. This brings additional flexibility to the language.

There are two basic types of functions. Built-in functions and user defined ones. The built-in functions are part of the Python language. Examples are: `dir()`, `len()`, or `abs()`. The user defined functions are functions created with the `def` keyword.

Defining functions

A function is created with the `def` keyword. The statements in the block of the function must be indented.

```
def function():
    pass
```

The `def` keyword is followed by the function name with round brackets and a colon. The indented statements form a *body* of the function.

The function is later executed when needed. We say that we *call* the function. If we call a function, the statements inside the function body are executed. They are not executed until the function is called.

```
myfunc()
```

To call a function, we specify the function name with the round brackets.

ret.py

```
#!/usr/bin/python

"""
The ret.py script shows how to work with
functions in Python.
Author: Jan Bodnar
ZetCode, 2016
"""

def showModuleName():

    print __doc__

def getModuleFile():

   return __file__
```

```
a = showModuleName()
b = getModuleFile()

print a, b
```

The string at the top of the script is called the documentation string. It documents the current script. The file in which we put Python code is called a *module*. We define two functions. The first function will print the module doc string. The second will `return` the path of our module. Function may or may not return a value. If they explicitly do not return a value, they implicitly return `None`. The `__doc__` and `__file__` are special state attributes. Note that there are two underscores on both sides of the attribute.

```
$ ./ret.py

The ret.py script shows how to work with
functions in Python.
Author: Jan Bodnar
ZetCode, 2016

None ./ret.py
```

This is the output of the program.

Definitions of functions must precede their usage. Otherwise the interpreter will complain with a `NameError`.

func_prec.py

```
#!/usr/bin/python

# func_prec.py

def f1():
    print "f1()"

f1()
#f2()

def f2():
    print "f2()"
```

In the above example, we have two definitions of functions. One line is commented. Function call cannot be ahead of its definition.

```
#f2()

def f2():
    print "f2()"
```

We can call the `f2()` only after its definition. Uncommenting the line we get a `NameError`.

Where to define functions

Functions can be defined inside a module, a class, or another function. Function defined inside a class is called a *method*.

defining.py

```
#!/usr/bin/python

# defining.py

class Some:

    @staticmethod
    def f():
        print "f() method"

def f():
    print "f() function"

def g():
    def f():
        print "f() inner function"
    f()
```

```
Some.f()
f()
g()
```

In this example, we define an `f()` function in three different places.

```
class Some:

    @staticmethod
    def f():
        print "f() method"
```

A static method is defined with a decorator in a `Some` class.

```
def f():
    print "f() function"
```

The function is defined in a module.

```
def g():
    def f():
        print "f() inner function"
    f()
```

Here the `f()` function is defined inside another `g()` function. It is an inner function.

```
Some.f()
f()
g()
```

The static method is called by specifying the class name, the dot operator and the function name with square brackets. Other functions are called using their names and square brackets.

```
$ ./defining.py
f() method
f() function
f() inner function
```

This is the output.

Functions are objects

Functions in Python are objects. They can be manipulated like other objects in Python. Therefore functions are called first-class citizens. This is not true i other OOP languages like Java or C#.

fun_obj.py

```
#!/usr/bin/python

# fun_obj.py

def f():
    """This function prints a message """
    print "Today it is a cloudy day"

print isinstance(f, object)
print id(f)

print f.func_doc
print f.func_name
```

In this script we show that our function is an object, too.

```
def f():
    """This function prints a message """
    print "Today it is a cloudy day"
```

We define an `f()` function. It prints a message to the console. It also has a documentation string.

```
print isinstance(f, object)
```

The `isinstance()` function checks whether the `f()` function is an instance of the `object`. All objects in Python inherit from this base entity.

```
print id(f)
```

Each object in Python has a unique id. The `id()` function returns the object's id.

```
print f.func_doc
print f.func_name
```

Objects may have attributes. Here we print two attributes of the function.

```
$ ./fun_obj.py
True
3077407212
This function prints a message
f
```

This is the output of the program.

Objects can be stored in collections and passed to functions.

fun_coll.py

```
#!/usr/bin/python

# fun_coll.py

def f():
    pass

def g():
    pass

def h(f):
    print id(f)

a = (f, g, h)

for i in a:
    print i

h(f)
h(g)
```

We define three functions. We place them in a tuple and pass them to a function.

```
a = (f, g, h)

for i in a:
    print i
```

We place three function objects in a tuple and traverse it with a for loop.

```
h(f)
h(g)
```

We pass the `f()` and `g()` functions to the `h()` function.

```
$ ./fun_coll.py
<function f at 0xb7664fb4>
<function g at 0xb766c1b4>
<function h at 0xb766c3ac>
3076935604
3076964788
```

This is the output of the `fun_coll` program.

Three kinds of functions

Looking from a particular point of view, we can discern three kinds of functions. Functions that are always available for usage, functions that are contain within external modules, which must be imported and functions defined by a programmer with the `def` keyword.

three_kinds.py

```
#!/usr/bin/python

from math import sqrt

def cube(x):
    return x * x * x

print abs(-1)
print cube(9)
print sqrt(81)
```

Three kinds of functions are present in the above code.

```
from math import sqrt
```

The `sqrt()` function is imported from the math module.

```
def cube(x):
    return x * x * x
```

The cube() function is a custom defined function.

```
print abs(-1)
```

The `abs()` function is a built-in function readily accessible. It is part of the core of the language.

The return keyword

A function is created to do a specific task. Often there is a result from such a task. The `return` keyword is used to return values from a function. A function may or may not return a value. If a function does not have a return keyword, it will send `None`.

returning.py

```
#!/usr/bin/python

# returning.py

def showMessage(msg):
    print msg

def cube(x):
    return x * x * x

x = cube(3)
print x

showMessage("Computation finished.")
print showMessage("Ready.")
```

We have two functions defined. One uses the `return` keyword, the other one does not.

```
def showMessage(msg):
    print msg
```

The `showMessage()` function does not return explicitly a value. It shows a message on the console.

```
def cube(x):
    return x * x * x
```

The cube() functions computes an expression and returns its result with the `return` keyword.

```
x = cube(3)
```

In this line we call the `cube()` function. The result of the computation of the `cube()` function is returned and assigned to the `x` variable. It holds the result value now.

```
showMessage("Computation finished.")
```

We call the `showMessage()` function with a message as a parameter. The message is printed to the console. We do not expect a value from this function.

```
print showMessage("Ready.")
```

This code produces two lines. One is a message printed by the `showMessage()` function. The other is the `None` value, which is implicitly sent by functions without the `return` statement.

```
$ ./returning.py
27
Computation finished.
Ready.
None
```

This is the example output.

We can send more that one value from a function. The objects after the `return` keyword are separated by commas.

returning2.py

```
#!/usr/bin/python

# returning2.py

n = [1, 2, 3, 4, 5]

def stats(x):

    mx = max(x)
    mn = min(x)
    ln = len(x)
    sm = sum(x)

    return mx, mn, ln, sm

mx, mn, ln, sm = stats(n)
print stats(n)

print mx, mn, ln, sm
```

There is a definition of a `stats()` function. This function returns four values.

```
return mx, mn, ln, sm
```

The `return` keyword sends back four numbers. The numbers are separated by a comma character. In fact, we have sent a tuple containing these four value
We could also return a list instead of a tuple.

```
mx, mn, ln, sm = stats(n)
```

The returned values are assigned to local variables.

```
$ ./returning2.py
(5, 1, 5, 15)
5 1 5 15
```

This is the output.

Function redefinition

Python is dynamic in nature. It is possible to redefine an already defined function.

redefinition.py

```
#!/usr/bin/python

# redefinition.py

from time import gmtime, strftime
```

```
def showMessage(msg):
    print msg

showMessage("Ready.")

def showMessage(msg):
    print strftime("%H:%M:%S", gmtime()),
    print msg

showMessage("Processing.")
```

We define a `showMessage()` function. Later we provide a new definition of the same function.

```
from time import gmtime, strftime
```

From the time module we import two functions which are used to compute the current time.

```
def showMessage(msg):
    print msg
```

This is the first definition of a function. It only prints a message to the console.

```
def showMessage(msg):
    print strftime("%H:%M:%S", gmtime()),
    print msg
```

Later in the source code, we set up a new definition of the `showMessage()` function. The message is preceded with a timestamp.

```
$ ./redefinition.py
Ready.
23:49:33 Processing.
```

This is the output.

Function arguments

Most functions accept arguments. Arguments are values that are sent to the function. The functions process the values and optionally return some value back.

fahrenheit.py

```
#!/usr/bin/python

# fahrenheit

def C2F(c):
    return c * 9/5 + 32

print C2F(100)
print C2F(0)
print C2F(30)
```

In our example, we convert Celsius temperature to Fahrenheit. The `C2F()` function accepts one argument c, which is the Celsius temperature.

```
$ ./fahrenheit.py
212
32
86
```

The arguments in Python functions may have implicit values. An implicit value is used if no value is provided.

fun_implicit.py

```
#!/usr/bin/python

# fun_implicit.py

def power(x, y=2):

    r = 1

    for i in range(y):
```

```
        r = r * x

    return r

print power(3)
print power(3, 3)
print power(5, 5)
```

Here we created a power function. The function has one argument with an implicit value. We can call the function with one or two arguments.

```
$ ./fun_implicit.py
9
27
3125
```

Python functions can specify their arguments with a keyword. This means that when calling a function, we specify both a keyword and a value. When we have multiple arguments and they are used without keywords, the order in which we pass those arguments is crucial. If we expect a name, age, or sex in a function without keywords, we cannot change their order. If we use keywords, we can.

fun_keywords.py

```
#!/usr/bin/python

# fun_keywords.py

def display(name, age, sex):

   print ("Name: ", name)
   print ("Age: ", age)
   print ("Sex: ", sex)


display("Lary", 43, "M")
display("Joan", 24, "F")
```

In this example, the order in which we specify the arguments is important. Otherwise, we get incorrect results.

```
$ ./fun_keywords.py
Name:  Lary
Age:  43
Sex:  M
Name:  Joan
Age:  24
Sex:  F
```

fun_keywords2.py

```
#!/usr/bin/python

# fun_keywords2.py

def display(name, age, sex):

   print "Name: ", name
   print "Age: ", age
   print "Sex: ", sex


display(age=43, name="Lary", sex="M")
display(name="Joan", age=24, sex="F")
```

Now we call the functions with their keywords. The order may be changed, although it is not recommended to do so. Note that we cannot use a non-keyword argument after a keyword argument. This would end in a syntax error.

```
display("Joan", sex="F", age=24)
```

This is a legal construct. A non-keyword argument may be followed by keyword arguments.

```
display(age=24, name="Joan", "F")
```

This will end in a syntax error. A non-keyword argument may not follow keyword arguments.

Functions in Python can even accept arbitrary number of arguments.

arbitrary_args.py

```
#!/usr/bin/python

# arbitrary_args.py

def sum(*args):
   '''Function returns the sum
   of all values'''

   r = 0

   for i in args:
      r += i

   return r


print sum.__doc__
print sum(1, 2, 3)
print sum(1, 2, 3, 4, 5)
```

We use the `*` operator to indicate that the function will accept arbitrary number of arguments. The `sum()` function will return the sum of all arguments. Th first string in the function body is called the function documentation string. It is used to document the function. The string must be in triple quotes.

```
$ ./arbitrary_args.py
Function returns the sum
   of all values
6
15
```

We can also use the `**` construct in our functions. In such a case, the function will accept a dictionary. The dictionary has arbitrary length. We can then normally parse the dictionary, as usual.

details.py

```
#!/usr/bin/python

# details.py

def display(**details):

   for i in details:
      print "%s: %s" % (i, details[i])


display(name="Lary", age=43, sex="M")
```

This example demonstrates such a case. We can provide arbitrary number of key-value arguments. The function will handle them all.

```
$ ./details.py
age: 43
name: Lary
sex: M
```

Passing by reference

Parameters to functions are passed by reference. Some languages pass copies of the objects to functions. Passing objects by reference has two important conclusions: a) the process is faster than if copies of objects were passed; b) mutable objects that are modified in functions are permanently changed.

passing_by_reference.py

```
#!/usr/bin/python

# passing_by_reference.py

n = [1, 2, 3, 4, 5]

print "Original list:", n

def f(x):
```

```
    x.pop()
    x.pop()
    x.insert(0, 0)
    print "Inside f():", x

f(n)

print "After function call:", n
```

In our example, we pass a list of integers to a function. The object is modified inside the body of the function. After calling the function, the original objec the list of integers is modified.

```
def f(x):

    x.pop()
    x.pop()
    x.insert(0, 0)
    print "Inside f():", x
```

In the body of the function we work with the original object. Not with a copy of the object. In many programming languages, we woud receive a copy of a object by default.

```
$ ./passing_by_reference.py
Original list: [1, 2, 3, 4, 5]
Inside f(): [0, 1, 2, 3]
After function call: [0, 1, 2, 3]
```

Once the list was modified it was modified for good.

Global and local variables

Next we will talk about how variables are used in Python functions.

local_variable.py

```
#!/usr/bin/python

# local_variable.py

name = "Jack"

def f():
   name = "Robert"
   print "Within function", name

print "Outside function", name
f()
```

A variable defined in a function body has a *local* scope. It is valid only within the body of the function.

```
$ ./local_variable.py
Outside function Jack
Within function Robert
```

This is example output.

global_variable.py

```
#!/usr/bin/python

# global_variable.py

name = "Jack"

def f():
   print "Within function", name

print "Outside function", name
f()
```

By default, we can get the contents of a *global variable* inside the body of a function. But if we want to change a global variable in a function, we must use the `global` keyword.

```
$ ./global_variable.py
Outside function Jack
Within function Jack
```

global_variable2.py

```
#!/usr/bin/python

# global_variable2.py

name = "Jack"

def f():
   global name
   name = "Robert"
   print "Within function", name


print "Outside function", name
f()
print "Outside function", name
```

Now, we will change the contents of a global name variable inside a function.

```
global name
name = "Robert"
```

Using the `global` keyword, we reference the variable defined outside the body of the function. The variable is given a new value.

```
$ ./global_variable2.py
Outside function Jack
Within function Robert
Outside function Robert
```

Anonymous functions

It is possible to create anonymous functions in Python. Anonymous functions do not have a name. With the `lambda` keyword, little anonymous functions can be created. Anonymous functions are also called lambda functions by Python programmers. They are part of the functional paradigm incorporated in Python.

Lambda functions are restricted to a single expression. They can be used wherever normal functions can be used.

lambda.py

```
#!/usr/bin/python

# lambda.py

y = 6

z = lambda x: x * y
print z(8)
```

This is a small example of the lambda function.

```
z = lambda x: x * y
```

The `lambda` keyword creates an anonymous function. The x is a parameter that is passed to the lambda function. The parameter is followed by a colon character. The code next to the colon is the expression that is executed, when the lambda function is called. The lambda function is assigned to the z variable.

```
print z(8)
```

The lambda function is executed. The number 8 is passed to the anonymous function and it returns 48 as the result. Note that z is not a name for this function. It is only a variable to which the anonymous function was assigned.

```
$ ./lambda.py
48
```

Output of the example.

The lambda function can be used elegantly with other functional parts of the Python language, like `map()` or `filter()` functions.

lambda2.py

```
#!/usr/bin/python

# lambda2.py

cs = [-10, 0, 15, 30, 40]

ft = map(lambda t: (9.0/5)*t + 32, cs)
print ft
```

In the example we have a list of Celsius temperatures. We create a new list containing temperatures in Fahrenheit.

```
ft = map(lambda t: (9.0/5)*t + 32, cs)
```

The `map()` function applies the anonymous function to each element of the `cs` list. It creates a new `ft` list containing the computed Fahrenheit temperature

```
$ ./lambda2.py
[14.0, 32.0, 59.0, 86.0, 104.0]
```

This is example output.

This chapter was about functions in Python.