Lecture 5 - Functions

Adnan Ferdous Ashrafi

Stamford University Bangladesh

Table of Contents

Definition

Function

In the context of programming, a *function* is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name.

Function calls

Let's start with an example:

```
>>> type(42)
"<class 'int'>"
```

The name of the function is type. The expression in parentheses is called the **argument** of the function. The result, for this function, is the type of the argument. It is common to say that a function *"takes"* an argument and *"returns"* a result. The result is also called the **return value**.
Python provides functions that convert values from one type to another. The *int* function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

Function calls

int can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

float converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finally, *str* converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Math Functions

Python has a math module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions.

Before we can use the functions in a module, we have to import it with an import statement:

```
>>> import math
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called **dot notation**.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio) #finds the logarithm
>>> radians = 0.7
>>> height = math.sin(radians) #finds the sine
```

Math Functions

The second example finds the sine of radians. The variable name radians is a hint that *sin* and the other trigonometric functions (*cos*, *tan*, etc.) take arguments in radians. To convert from degrees to radians, divide by 180 and multiply by π:

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

The expression *math.pi* gets the variable π from the math module. Its value is a floating point approximation of π, accurate to about 15 digits.

If you know trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

Composition

Definition

One of the most useful features of programming languages is their ability to take small building blocks and **compose** i.e. combine them together. For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
1 x = math.sin(degrees / 360.0 * 2 * math.pi)
```

And even function calls:

```
1 x = math.exp(math.log(x+1))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error.

```
1 >>> minutes = hours * 60 # right
2 >>> hours * 60 = minutes # wrong!
3 "SyntaxError: can't assign to operator"
```

Adding new functions

Definition

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A **function definition** specifies the name of a new function and the sequence of statements that run when the function is called.

Here's an example:

```
1 def print_lyrics():
2     print("I'm a computer scientist, and I'm okay.")
3     print("I work all night and I sleep all day.")
```

Syntax of a Function

def is a keyword that indicates that this is a function definition. The name of the function is *print_lyrics*. The rules for function names are the same as for variable names: letters, numbers and underscore are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments.

The first line of the function definition is called the **header**; the rest is called the **body**. The header has to end with a colon and the body has to be indented. By convention, indentation is always four spaces. The body can contain any number of statements.

Syntax of a Function

Defining a function creates a function object, which has type function:

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
"<class 'function'>"
```

The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
"I'm a computer scientist, and I'm okay."
"I work all night and I sleep all day."
```

Syntax of a Function

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called *repeat_lyrics*:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

And then call *repeat_lyrics*:

```
>>> repeat_lyrics()
"I'm a computer scientist, and I'm okay."
"I work all night and I sleep all day."
"I'm a computer scientist, and I'm okay."
"I work all night and I sleep all day."
```

Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
1  def print_lyrics():
2      print("I'm a computer scientist, and I'm okay.")
3      print("I work all night and I sleep all day.")
4
5  def repeat_lyrics():
6      print_lyrics()
7      print_lyrics()
8
9  repeat_lyrics()
```

This program contains two function definitions: *print_lyrics* and *repeat_lyrics*. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not run until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can run it. In other words, the function definition has to run before the function gets called.

Flow of Execution

Definition

To ensure that a function is defined before its first use, you have to know the order statements run in, which is called the flow of execution.

Execution always begins at the first statement of the program. Statements are run one at a time, in order from top to bottom. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function don't run until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off.

Flow of Execution

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to run the statements in another function. Then, while running that new function, the program might have to run yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

In summary, when you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

Parameters and arguments

Some of the functions we have seen require arguments. For example, when you call *math.sin* you pass a number as an argument. Some functions take more than one argument: *math.pow* takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called parameters. Here is a definition for a function that takes an argument:

```
def print_twice(value):
    print(value)
    print(value)
```

This function assigns the argument to a parameter named *value*. When the function is called, it prints the value of the parameter (whatever it is) twice.

Parameters and arguments

This function works with any value that can be printed.

```
>>> print_twice('Stamford')
Stamford
Stamford
>>> print_twice(42)
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

The same rules of composition that apply to built-in functions also apply to programmer defined functions, so we can use any kind of expression as an argument for *print_twice*:

Parameters and arguments

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions *'Spam '*4* and *math.cos(math.pi)* are only evaluated once.
You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (michael) has nothing to do with the name of the parameter (*value*). It doesn't matter what the value was called back home (in the caller); here in print_twice, we call everybody *value*.

Variables and parameters are local

When you create a variable inside a function, it is local, which means that it only exists inside the function. For example:

```
def message_twice(part1, part2):
    message = part1 + part2
    print_twice(message)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```
>>> line1 = 'Hello '
>>> line2 = 'World'
>>> message_twice(line1, line2)
Hello World.
Hello World.
```

Variables and parameters are local

When *message_twice* terminates, the variable *message* is destroyed. If we try to print it, we get an exception:

```
>>> print(message)
"NameError: name 'message' is not defined"
```

Parameters are also local. For example, outside *print_twice*, there is no such thing as *value*.

Fruitful functions and void functions

Some of the functions we have used, such as the math functions, return results and so can be called as **fruitful functions**. Other functions, like *print_twice*, perform an action but don't return a value. They are called **void functions**.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.2360679774997898
```

Fruitful functions and void functions

But in a script, if you call a fruitful function all by itself, the return value is lost forever!

```
math.sqrt(5)
# this will not give out any value
```

This script computes the square root of 5, but since it doesn't store or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you assign the result to a variable, you get a special value called *None*.

```
>>> result = print_twice('Hello')
Hello
Hello
>>> print(result)
None
```

Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

1. Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
2. Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
3. Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
4. Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Adding two numbers

In order to add two numbers using a function, we can define a function named *add_num* with two parameters, say *x* and *y*. Inside the function we can add the two numbers and finally return the value to use the result to display.

```
1  def add_num(x , y):
2      res = x + y
3      return res
4
5  print("Let's see the function in action!")
6  result = add_num(100,25)
7  print("The addition of 100 and 25 is", result)
```

The output is as follows:

```
Let's see the function in action!
The addition of 100 and 25 is 125
```

Finding the summation of a series

In order to find the summation of a series, from a *first_num* to a *second_num* with an increment of *inc* we can define a function name *series_sum* with three parameters, say *num_1*, *num_2* and *inc*. Inside the function we can run a loop that traverses from *first_num* to *second_num* with an increment of *inc* in each iteration and adds the numbers each time. Finally we can return the result to the program for displaying or using it in any other cases.

```
1  def series_sum(num_1, num_2, inc):
2      sum = 0
3      current_num = num_1
4      while(current_num <= num_2):
5          sum = sum + current_num
6          current_num += inc
7      return sum
8  # Let's sum the series 1,3,5,7,9,.....,1235
9  # The first_num = 1, second_num = 1235 and increment inc is 2
10 result = series_sum(1, 1235, 2)
11 print("The summation of the series from 1 to 1235 is: ", result)
```

The output is as follows:

```
The summation of the series from 1 to 1235 is:  381924
```

Solving an equation

Suppose we have a quadratic equation as follows:

$$ax^2 + bx + c = 0$$

In order to solve the equation for x, we can use this familiar formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

So, in order to find the value of x we can simply plug in the values of a, b and c, and solve the equation for x. We can design a simple function that will take in the values of a, b and c and return the result of x.

Solving an equation

```
1  import cmath # As the result is complex, we imported complex math module
2
3  def solve_x(a,b,c):
4      numerator = -b + cmath.sqrt(b**2 - 4*a*c)
5      denominator = 2*a
6      x_value = numerator-denominator
7
8      return x_value
9
10 # Suppsoe we have an equation 7x^2 - 8x + 3 = 0
11 # Then a = 7, b = -8 and c = 3
12 a, b, c = 7, -8, 3
13 x = solve_x(a, b, c)
14 print("The value of x is: ", x, "or", -x)
```

The output is as follows:

```
The value of x is:  (-6+4.47213595499958j) or (6-4.47213595499958j)
```

Glossary

function: A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function definition: A statement that creates a new function, specifying its name, parameters, and the statements it contains.

function object: A value created by a function definition. The name of the function is a variable that refers to a function object.

header: The first line of a function definition.

body: The sequence of statements inside a function definition.

parameter: A name used inside a function to refer to the value passed as an argument.

function call: A statement that runs a function. It consists of the function name followed by an argument list in parentheses.

Glossary

argument: A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

local variable: A variable defined inside a function. A local variable can only be used inside its function.

return value: The result of a function. If a function call is used as an expression, the return value is the value of the expression.

fruitful function: A function that returns a value.

void function: A function that always returns None.

None: A special value returned by void functions.

module: A file that contains a collection of related functions and other definitions.

Glossary

import statement: A statement that reads a module file and creates a module object.

module object: A value created by an import statement that provides access to the values defined in a module.

dot notation: The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

composition: Using an expression as part of a larger expression, or a statement as part of a larger statement.

flow of execution: The order statements run in.

traceback: A list of the functions that are executing, printed when an exception occurs.

Further Reading

- Chapter 3 of Think Python(2nd Edition) - Allen B. Downey
- Chapter 8 of Python Crash Course - Eric Matthes
- Python Official Website
- Lecture 5 Jupyter Notebook

Thank you.
Any Questions?