**Chapter 4**

Functions

# Function calls

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. We have already seen one example of a *function call*:

>>> type(32)

<**class** 'int'>

The name of the function is type. The expression in parentheses is called the *argument* of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the type 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 called the *return value*.

# Built-in functions

Python provides a number of important built-in functions that we can use without needing to provide the function definition. The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.

The max and min functions give us the largest and smallest values in a list, respec- tively:

>>> max('Hello world')

*’w’*

>>> min('Hello world')

*’ ’*

>>>

43

The max function tells us the “largest character” in the string (which turns out to be the letter “w”) and the min function shows us the smallest character (which turns out to be a space).

Another very common built-in function is the len function which tells us how many items are in its argument. If the argument to len is a string, it returns the number of characters in the string.

>>> len('Hello world') 11

>>>

These functions are not limited to looking at strings. They can operate on any set of values, as we will see in later chapters.

You should treat the names of built-in functions as reserved words (i.e., avoid using “max” as a variable name).

# Type conversion functions

Python also provides built-in functions that convert values from one type to an- other. 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() **with** base 10: 'Hello'

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)

# Math functions

| math.cos() | Returns the cosine of a number |
| --- | --- |
| math.degrees() | Converts an angle from radians to degrees |
| math.floor() | Rounds a number down to the nearest integer |
| math.log() | Returns the natural logarithm of a number, or the logarithm of number to base |
| math.pow() | Returns the value of x to the power of y |
| math.tan() | Returns the tangent of a number |

Python has a math module that provides most of the familiar mathematical func- tions. Before we can use the module, we have to import it:

>>> import math

This statement creates a *module object* named math. If you print the module object, you get some information about it:

>>> print(math)

<module 'math' (built-in)>

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)

>>> radians = 0.7

>>> height = math.sin(radians)

The first example computes the logarithm base 10 of the signal-to-noise ratio. The math module also provides a function called log that computes logarithms base e.

The second example finds the sine of radians. The name of the variable 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 360 and multiply by 2π:

>>> degrees = 45

>>> radians = degrees / 360.0 \* 2 \* math.pi

>>> math.sin(radians) 0.7071067811865476

The expression math.pi gets the variable pi from the math module. The value of this variable is an approximation of π, accurate to about 15 digits.

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

>>> math.sqrt(2) / 2.0 0.7071067811865476

# Random numbers

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be *deterministic*. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applica- tions, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to use *al- gorithms* that generate *pseudorandom* numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The random module provides functions that generate pseudorandom numbers (which I will simply call “random” from here on).

The function random returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call random, you get the next number in a long series. To see a sample, run this loop:

import random

**for** i in range(10):

x = random.random() print(x)

This program produces the following list of 10 random numbers between 0.0 and up to but not including 1.0.

0.11132867921152356

0.5950949227890241

0.04820265884996877

0.841003109276478

0.997914947094958

0.04842330803368111

0.7416295948208405

0.510535245390327

0.27447040171978143

0.028511805472785867

## Exercise 1: Run the program on your system and see what numbers you get. Run the program more than once and see what numbers you get.

The random function is only one of many functions that handle random numbers. The function randint takes the parameters low and high, and returns an integer between low and high (including both).

>>> random.randint(5, 10)

5

>>> random.randint(5, 10)

To choose an element from a sequence at random, you can use choice:

>>> t = [1, 2, 3]

>>> random.choice(t) 2

>>> random.choice(t) 3

The random module also provides functions to generate random values from con- tinuous distributions including Gaussian, exponential, gamma, and a few more.

# 4.6 Adding new functions

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 execute when the function is called. Once we define a function, we can reuse the function over and over throughout our program.

Here is an example:

**def** print\_lyrics():

print("I'm a lumberjack, and I'm okay.") print('I sleep all night and I work all day.')

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 some punctuation marks 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. Later we will build functions that take arguments as their inputs.

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, the indentation is always four spaces. The body can contain any number of statements.

If you type a function definition in interactive mode, the interpreter prints ellipses (*. . .* ) to let you know that the definition isn’t complete:

>>> **def** print\_lyrics():

... print("I'm a lumberjack, and I'm okay.")

... print('I sleep all night and I work all day.')

...

To end the function, you have to enter an empty line (this is not necessary in a script).

Defining a function creates a variable with the same name.

>>> print(print\_lyrics)

<function print\_lyrics at 0xb7e99e9c>

>>> print(type(print\_lyrics))

<**class** 'function'>

The value of print\_lyrics is a *function object*, which has type “function”. The syntax for calling the new function is the same as for built-in functions:

>>> print\_lyrics()

I'm a lumberjack, and I'm okay.

I sleep all night and I work all day.

Once you have defined a function, you can use it inside another function. For exam- ple, 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 lumberjack, and I'm okay.

I sleep all night and I work all day. I'm a lumberjack, and I'm okay.

I sleep all night and I work all day.

But that’s not really how the song goes.

# 4.7 Definitions and uses

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

**def** print\_lyrics():

print("I'm a lumberjack, and I'm okay.") print('I sleep all night and I work all day.')

**def** repeat\_lyrics(): print\_lyrics() print\_lyrics()

repeat\_lyrics()

*# Code:* [*http://www.py4e.com/code3/lyrics.p*](http://www.py4e.com/code3/lyrics.py)

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 get executed 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 execute it. In other words, the function definition has to be executed before the first time it is called.

## Exercise 2: Move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

**Exercise 3: Move the function call back to the bottom and move the definition of print\_lyrics after the definition of repeat\_lyrics. What happens when you run this program?**

**4.8 Flow of execution**

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

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function *definitions* do not alter the flow of execution of the program, but re- member that statements inside the function are not executed 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, executes all the statements there, and then comes back to pick up where it left off.

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 execute the state- ments in another function. But while executing that new function, the program might have to execute 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.

What’s the moral of this sordid tale? 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.

# 4.9 Parameters and arguments

Some of the built-in 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 an example of a user-defined function that takes an argument:

**def** print\_twice(bruce): print(bruce) print(bruce)

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

This function works with any value that can be printed.

>>> print\_twice('Spam') Spam

Spam

>>> print\_twice(17) 17

17

>>> import math

>>> print\_twice(math.pi) 3.141592653589793

3.141592653589793

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for print\_twice:

>>> 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 (bruce). It doesn’t matter what the value was called back home (in the caller); here in print\_twice, we call everybody bruce

# 4.10 Fruitful functions and void functions

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them *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) 2.23606797749979

But in a script, if you call a fruitful function and do not store the result of the function in a variable, the return value vanishes into the mist!

math.sqrt(5)

This script computes the square root of 5, but since it doesn’t store the result in a variable 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 try to assign the result to a variable, you get a special value called None.

>>> result = print\_twice('Bing') Bing

Bing

>>> print(result) None

The value None is not the same as the string “None”. It is a special value that has its own type:

>>> print(type(None))

<**class** 'NoneType'>

To return a result from a function, we use the return statement in our function. For example, we could make a very simple function called add two that adds two numbers together and returns a result

**def** addtwo(a, b): added = a + b **return** added

x = addtwo(3, 5) print(x)

*# Code:* [*http://www.py4e.com/code3/addtwo.py*](http://www.py4e.com/code3/addtwo.py)

When this script executes, the print statement will print out “8” because the add two function was called with 3 and 5 as arguments. Within the function, the parameters a and b were 3 and 5 respectively. The function computed the sum of the two numbers and placed it in the local function variable named added. Then it used the return statement to send the computed value back to the calling code as the function result, which was assigned to the variable x and printed out.

# 4.11 Why functions?

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

* Creating a new function gives you an opportunity to name a group of state- ments, which makes your program easier to read, understand, and debug.
* 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.
* Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
* Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Throughout the rest of the book, often we will use a function definition to explain a concept. Part of the skill of creating and using functions is to have a function properly capture an idea such as “find the smallest value in a list of values”. Later we will show you code that finds the smallest in a list of values and we will present it to you as a function named min which takes a list of values as its argument and returns the smallest value in the list.

# 4.12 Debugging

If you are using a text editor to write your scripts, you might run into problems with spaces and tabs. The best way to avoid these problems is to use spaces exclusively (no tabs). Most text editors that know about Python do this by default, but some don’t.

Tabs and spaces are usually invisible, which makes them hard to debug, so try to find an editor that manages indentation for you

Also, don’t forget to save your program before you run it. Some development environments do this automatically, but some don’t. In that case, the program you are looking at in the text editor is not the same as the program you are running.

Debugging can take a long time if you keep running the same incorrect program over and over!

Make sure that the code you are looking at is the code you are running. If you’re not sure, put something like print("hello") at the beginning of the program and run it again. If you don’t see hello, you’re not running the right program!

# 4.13 Glossary

**algorithm** A general process for solving a category of problems.

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

**body** The sequence of statements inside a function definition.

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

**deterministic** Pertaining to a program that does the same thing each time it runs, given the same inputs.

**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.

**flow of execution** The order in which statements are executed during a program run.

**fruitful function** A function that returns a value.

**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 call** A statement that executes a function. It consists of the function name followed by an argument list.

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

**function object** A value created by a function definition. The name of the func- tion is a variable that refers to a function object.

**header** The first line of a function definition.

**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 data and code defined in a module.

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

**pseudorandom** Pertaining to a sequence of numbers that appear to be random, but are generated by a deterministic program.

**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.

**void function** A function that does not return a value.

# 4.14 Exercises

## Exercise 4: What is the purpose of the “def” keyword in Python?

1. It is slang that means “the following code is really cool”
2. It indicates the start of a function
3. It indicates that the following indented section of code is to be stored for later
4. b and c are both true
5. None of the above

## Exercise 5: What will the following Python program print out?

**def** fred(): print("Zap")

**def** jane(): print("ABC")

jane()

fred()

jane()

1. Zap ABC jane fred jane
2. Zap ABC Zap
3. ABC Zap jane
4. ABC Zap ABC
5. Zap Zap Zap

## Exercise 6: Rewrite your pay computation with time-and-a-half for over- time and create a function called computepay which takes two parameters (hours and rate).

Enter Hours: 45

Enter Rate: 10

Pay: 475.0

## Exercise 7: Rewrite the grade program from the previous chapter using a function called computegrade that takes a score as its parameter and returns a grade as a string.

Score Grade

>= 0.9 A

>= 0.8 B

>= 0.7 C

>= 0.6 D

< 0.6 F

Enter score: 0.95 A

Enter score: perfect Bad scor

Enter score: 10.0 Bad score

Enter score: 0.75 C

Enter score: 0.5 F

Run the program repeatedly to test the various different values for input.