12.2. Function Definition

The syntax for creating a named function, a **function definition**, is:

**def** name( parameters ):

statements

You can make up any names you want for the functions you create, except that you can’t use a name that is a Python keyword, and the names must follow the rules for legal identifiers that were given previously. The parameters specify what information, if any, you have to provide in order to use the new function. Another way to say this is that the parameters specify what the function needs to do its work.

There can be any number of statements inside the function, but they have to be indented from the def. In the examples in this book, we will use the standard indentation of four spaces. Function definitions are the third of several **compound statements** we will see, all of which have the same pattern:

1. A header line which begins with a keyword and ends with a colon.
2. A **body** consisting of one or more Python statements, each indented the same amount – *4 spaces is the Python standard* – from the header line.

We’ve already seen the for statement which has the same structure, with an indented block of code, and the if, elif, and else statements that do so as well.

In a function definition, the keyword in the header is def, which is followed by the name of the function and some *parameter names* enclosed in parentheses. The parameter list may be empty, or it may contain any number of parameters separated from one another by commas. In either case, the parentheses are required.

We will come back to the parameters in a little while, but first let’s see what happens when a function is executed, using a function without any parameters to illustrate.

Here’s the definition of a simple function, hello.

def hello():

"""This function says hello and greets you"""

print("Hello")

print("Glad to meet you")

​

**docstrings**

If the first thing after the function header is a string (some tools insist that it must be a triple-quoted string), it is called a **docstring** and gets special treatment in Python and in some of the programming tools.

Another way to retrieve this information is to use the interactive interpreter, and enter the expression <function\_name>.\_\_doc\_\_, which will retrieve the docstring for the function. So the string you write as documentation at the start of a function is retrievable by python tools *at runtime*. This is different from comments in your code, which are completely eliminated when the program is parsed.

By convention, Python programmers use docstrings for the key documentation of their functions.

# 12.3. Function Invocation

Defining a new function does not make the function run. To execute the function, we need a **function call**. This is also known as a **function invocation**.

**Note**

This section is a review of something we learned in the beginning of the textbook.

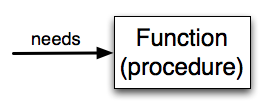
The way to invoke a function is to refer to it by name, followed by parentheses. Since there are no parameters for the function hello, we won’t need to put anything inside the parentheses when we call it. Once we’ve defined a function, we can call it as often as we like and its statements will be executed each time we call it.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Python 3.3   |  |  |  | | --- | --- | --- | |  | 1 | def hello(): | | 2 | print("Hello") | | 3 | print("Glad to meet you") | | 4 |  | | 5 | print(type(hello)) | | 6 | print(type("hello")) | | 7 |  | | 8 | hello() | | 9 | print("Hey, that just printed two lines with one line of code!") | | 10 | hello()  # do it again, just because we can... | |

# 12.4. Function Parameters

Named functions are nice because, once they are defined and we understand what they do, we can refer to them by name and not think too much about what they do. With parameters, functions are even more powerful, because they can do pretty much the same thing on each invocation, but not exactly the same thing. The parameters can cause them to do something a little different.

The figure below shows this relationship. A function needs certain information to do its work. These values, often called **arguments** or **actual parameters** or **parameter values**, are passed to the function by the user.



This type of diagram is often called a **black-box diagram** because it only states the requirements from the perspective of the user (well, the programmer, but the programmer who uses the function, who may be different than the programmer who created the function). The user must know the name of the function and what arguments need to be passed. The details of how the function works are hidden inside the “black-box”.

You have already been making function invocations with parameters. For example, when you write len("abc") or len([3, 9, "hello"]), len is the name of a function, and the value that you put inside the parentheses, the string “abc” or the list [3, 9, “hello”], is a parameter value.

When a function has one or more parameters, the names of the parameters appear in the function definition, and the values to assign to those parameters appear inside the parentheses of the function invocation. Let’s look at each of those a little more carefully.

In the definition, the parameter list is sometimes referred to as the **formal parameters** or **parameter names**. These names can be any valid variable name. If there is more than one, they are separated by commas.

In the function invocation, inside the parentheses one value should be provided for each of the parameter names. These values are separated by commas. The values can be specified either directly, or by any python expression including a reference to some other variable name.

That can get kind of confusing, so let’s start by looking at a function with just one parameter. The revised hello function personalizes the greeting: the person to greet is specified by the parameter.

**Check your understanding**

func-3-1: Which of the following is a valid function header (first line of a function definition)?

Top of Form

A. def greet(t):  
B. def greet:  
C. greet(t, n):  
D. def greet(t, n)

Bottom of Form

func-3-2: What is the name of the following function?

**def** print\_many(x, y):

*"""Print out string x, y times."""*

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

**print**(x)

Top of Form

A. def print\_many(x, y):  
B. print\_many  
C. print\_many(x, y)  
D. Print out string x, y times.

Bottom of Form

func-3-3: What are the parameters of the following function?

**def** print\_many(x, y):

*"""Print out string x, y times."""*

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

**print**(x)

Top of Form

A. i  
B. x  
C. x, y  
D. x, y, i

Bottom of Form

func-3-4: Considering the function below, which of the following statements correctly invokes, or calls, this function (i.e., causes it to run)?

**def** print\_many(x, y):

*"""Print out string x, y times."""*

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

**print**(x)

z = 3

Top of Form

A. print\_many(x, y)  
B. print\_many  
C. print\_many("Greetings")  
D. print\_many("Greetings", 10):  
E. print\_many("Greetings", z)

Bottom of Form

func-3-5: True or false: A function can be called several times by placing a function call in the body of a for loop.

Top of Form

A. True  
B. False

Bottom of Form

func-3-6: What output will the following code produce?

**def** cyu(s1, s2):

**if** len(s1) > len(s2):

**print**(s1)

**else**:

**print**(s2)

cyu("Hello", "Goodbye")

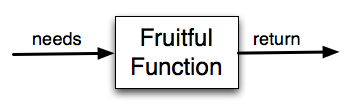
Top of Form

A. Hello  
B. Goodbye  
C. s1  
D. s2

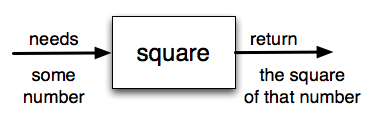
# 12.5. Returning a value from a function

Not only can you pass a parameter value into a function, a function can also produce a value. You have already seen this in some previous functions that you have used. For example, len takes a list or string as a parameter value and returns a number, the length of that list or string. range takes an integer as a parameter value and returns a list containing all the numbers from 0 up to that parameter value.

Functions that return values are sometimes called **fruitful functions**. In many other languages, a function that doesn’t return a value is called a **procedure**, but we will stick here with the Python way of also calling it a function, or if we want to stress it, a non-fruitful function.



How do we write our own fruitful function? Let’s start by creating a very simple mathematical function that we will call square. The square function will take one number as a parameter and return the result of squaring that number. Here is the black-box diagram with the Python code following.



def square(x):

y = x \* x

return y

​

toSquare = 10

result = square(toSquare)

print("The result of {} squared is {}.".format(toSquare, result))

​

The **return** statement is followed by an expression which is evaluated. Its result is returned to the caller as the “fruit” of calling this function. Because the return statement can contain any Python expression we could have avoided creating the **temporary variable** y and simply used return x\*x. Try modifying the square function above to see that this works just the same. On the other hand, using **temporary variables** like y in the program above makes debugging easier. These temporary variables are referred to as **local variables**.

# 12.6. A function that accumulates

We have used the len function a lot already. If it weren’t part of python, our lives as programmers would have been a lot harder.

Well, actually, not that much harder. Now that we know how to define functions, we could define len ourselves if it did not exist. Previously, we have used the accumlator pattern to count the number of lines in a file. Let’s use that same idea and just wrap it in a function definition. We’ll call it mylen to distinguish it from the real len which already exists. We actually could call it len, but that wouldn’t be a very good idea, because it would replace the original len function, and our implementation may not be a very good one.

def mylen(seq):

c = 0 # initialize count variable to 0

for \_ in seq:

c = c + 1 # increment the counter for each item in seq

return c

​

print(mylen("hello"))

print(mylen([1, 2, 7]))

# 12.7. Variables and parameters are local

An assignment statement in a function creates a **local variable** for the variable on the left hand side of the assignment operator. It is called local because this variable only exists inside the function and you cannot use it outside. For example, consider again the square function:

def square(x):

y = x \* x

return y

​

z = square(10)

print(y)

​

Try running this in Codelens. When a function is invoked in Codelens, the local scope is separated from global scope by a blue box. Variables in the local scope will be placed in the blue box while global variables will stay in the global frame. If you press the ‘last >>’ button you will see an error message. When we try to use y on line 6 (outside the function) Python looks for a global variable named y but does not find one. This results in the error: Name Error: 'y' is not defined.

The variable y only exists while the function is being executed — we call this its **lifetime**. When the execution of the function terminates (returns), the local variables are destroyed. Codelens helps you visualize this because the local variables disappear after the function returns. Go back and step through the statements paying particular attention to the variables that are created when the function is called. Note when they are subsequently destroyed as the function returns.

Formal parameters are also local and act like local variables. For example, the lifetime of x begins when square is called, and its lifetime ends when the function completes its execution.

So it is not possible for a function to set some local variable to a value, complete its execution, and then when it is called again next time, recover the local variable. Each call of the function creates new local variables, and their lifetimes expire when the function returns to the caller.

**Check Your Understanding**

func-7-1: True or False: Local variables can be referenced outside of the function they were defined in.

Top of Form

A. True  
B. False

Bottom of Form

func-7-2: Which of the following are local variables? Please, write them in order of what line they are on in the code.

numbers = [1, 12, 13, 4]

**def** foo(bar):

aug = str(bar) + "street"

**return** aug

addresses = []

**for** item **in** numbers:

addresses.append(foo(item))

func-7-3: What is the result of the following code?

**def** adding(x):

y = 3

z = y + x + x

**return** z

**def** producing(x):

z = x \* y

**return** z

**print**(producing(adding(4)))

Top of Form

A. 33  
B. 12  
C. There is an error in the code.

# 12.8. Global Variables

Variable names that are at the top-level, not inside any function definition, are called global.

It is legal for a function to access a global variable. However, this is considered **bad form** by nearly all programmers and should be avoided. This subsection includes some examples that illustrate the potential interactions of global and local variables. These will help you understand exactly how python works. Hopefully, they will also convince you that things can get pretty confusing when you mix local and global variables, and that you really shouldn’t do it.

Look at the following, nonsensical variation of the square function.

def badsquare(x):

y = x \*\* power

return y

​

power = 2

result = badsquare(10)

print(result)

​

Although the badsquare function works, it is silly and poorly written. We have done it here to illustrate an important rule about how variables are looked up in Python. First, Python looks at the variables that are defined as local variables in the function. We call this the **local scope**. If the variable name is not found in the local scope, then Python looks at the global variables, or **global scope**. This is exactly the case illustrated in the code above. power is not found locally in badsquare but it does exist globally. The appropriate way to write this function would be to pass power as a parameter. For practice, you should rewrite the badsquare example to have a second parameter called power.

# 12.9. Flow of Execution Summary

When you are working with functions it is really important to know the order in which statements are executed. This is called the **flow of execution** and we’ve already talked about it a number of times in this chapter.

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 remember that statements inside the function are not executed until the function is called. Function calls are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called 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 statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, the Python interperter is adept 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 does all that mean for us when we try to understand a program? Don’t read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the body of the function until you reach a point where that function is called.

**Check your understanding**

func-10-1: Consider the following Python code. Note that line numbers are included on the left.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **def** pow(b, p):  y = b \*\* p  **return** y  **def** square(x):  a = pow(x, 2)  **return** a  n = 5  result = square(n)  **print**(result) |

What does this function print?

Top of Form

A. 25  
B. 5  
C. 125  
D. 32

12.10. Glossary

**argument**

A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function. The argument can be the result of an expression which may involve operators, operands and calls to other fruitful functions.

**body**

The second part of a compound statement. The body consists of a sequence of statements all indented the same amount from the beginning of the header. The standard amount of indentation used within the Python community is 4 spaces.

**calling stack**

A sequence (stack) of frames, showing all the function calls that are in process but not yet complete. When one function’s code invokes another function call, there will be more than one frame on the stack.

**compound statement**

A statement that consists of two parts:

1. header - which begins with a keyword determining the statement type, and ends with a colon.
2. body - containing one or more statements indented the same amount from the header.

The syntax of a compound statement looks like this:

keyword expression:

statement

statement

...

**docstring**

If the first thing in a function body is a string (or, we’ll see later, in other situations too) that is attached to the function as its \_\_doc\_\_ attribute.

**flow of execution**

The order in which statements are executed during a program run.

**function**

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

**function call**

A statement that executes a function. It consists of the name of the function followed by a list of arguments enclosed in parentheses.

**function composition**

Using the output from one function call as the input to another.

**function definition**

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

**fruitful function**

A function that returns a value when it is called.

**global variable**

A variable defined at the top level, not inside any function.

**header line**

The first part of a compound statement. A header line begins with a keyword and ends with a colon (:)

**lifetime**

Variables and objects have lifetimes — they are created at some point during program execution, and will be destroyed at some time. In python, objects live as long as there is some variable pointing to it, or it is part of some other compound object, like a list or a dictionary. In python, local variables live only until the function finishes execution.

**local variable**

A variable defined inside a function. A local variable can only be used inside its function. Parameters of a function are also a special kind of local variable.

**method**

A special kind of function that is invoked on objects of particular types of objects, using the syntax <expr>.<methodname>(<additional parameter values>)

**None**

A special Python value. One use in Python is that it is returned by functions that do not execute a return statement with a return argument.

**parameter**

A name used inside a function to refer to the value which was passed to it as an argument.

**return value**

The value provided as the result of a function call.

# 12.11. Exercises

1. Write a function named num\_test that takes a number as input. If the number is greater than 10, the function should return “Greater than 10.” If the number is less than 10, the function should return “Less than 10.” If the number is equal to 10, the function should return “Equal to 10.”
2. Write a function that will return the number of digits in an integer.
3. Write a function that reverses its string argument.
4. Write a function that mirrors its string argument, generating a string containing the original string and the string backwards.
5. Write a function that removes all occurrences of a given letter from a string.
6. Although Python provides us with many list methods, it is good practice and very instructive to think about how they are implemented. Implement a Python function that works like the following:
   * + count
     + in
     + reverse
     + index
     + insert

Bottom of Form

Bottom of Form

Bottom of Form