4.2 Function definitions – the basics

There are three questions you must answer before you can write a function definition:

- What will the function do?
- What inputs does the function need?
- What, if anything, will the function return?

The answers to all of these should be clear from the way you use the function. It is often easiest to write the function call before you write its definition. This might seem backwards the definitions usually appear before the calls in the program code. However, modern editors make it easy to move around in the code; there is no reason that we need to write the program in the same order in which it is read.

A function definition has the form:

```
def <name>( <parameters> ):
    <body>
```

In other words, the definition

- starts with the word **def**;
- has the name of the function. This should be a unique name in your program. Python will allow you to give a function the same name as a variable, but the result might not work the way you expect. To emphasize the difference, some people capitalize the first letter of function names and have variable names start with lower-case letters;
- has a list of parameters inside parentheses. There might not be any parameters, but there must be parentheses in any case;
- has a body indented underneath the function header. This body is the code that is executed when the function is called.

There are a number of issues with parameters, so we will start with functions that have no parameters. One way to get data without inputs to the function is to ask the user for it. Here is a function that asks the user to input a string and prints it back out:

```
def ReadAndReport():
    text = input( "Enter a string: " )
    print( "You typed '%s'." % text )
```

The name of this function is ReadAndReport. It has no parameters. When it is called the two lines of the body: a **raw_input** statement and a **print** statement, will be executed. There must be a match between the parameters of a function and the arguments with which it is called. Since this function has no parameters, we call it with no arguments, as in

```
ReadAndReport()
```

Here is a complete program that makes use of function ReadAndReport():

```
def ReadAndReport():
    text = input( "Enter a string: " )
    print( "You typed '%s'." % text )

def main():
    ReadAndReport()
    ReadAndReport()

main()
```

Program 4.2.1: Coding with a function

The last line of this program, outside any of the definitions, calls function main(), so this is where execution starts. Inside function main(), function ReadAndReport() is called twice. Here is a typical run of this program:

Enter a string: `blah`
You typed 'blah'.
Enter a string: `La de dah`
You typed 'La de dah'.

It might be nice to call function ReadAndReport() in a loop, but this function doesn't give any way to signal the loop to quit. We need to return something if we want to exit the loop based on the input. Here is another version of our program that does this:

```
def ReadReportAndSignal():
    text = input( "Enter a string: " )
    if text == "":
        return "No input"
    else:
        print( "You typed '%s'." % text )
        return "Got input"

def main():
    done = False
    while not done:
        response = ReadReportAndSignal()
        if response == "No input":
            done = True
main()
```

Program 4.2.2: Returning a value from a function

Our function ReadReportAndSignal() still reads a string from the user and prints out a message about it. Unlike function ReadAndReport(), the new function returns a message about the string it has read: either the message `"No input"` or the message `"Got input"`. Function main() saves this message in variable response and uses this to decide when to exit from the **while**-loop.

The syntax of a **return**-statement is

```
return <expression>
```

When this is executed the expression is evaluated and control immediately goes back to the point where the function was called and the value of the expression is used as the value of the function call. If the expression is omitted and the statement is just

```
return
```

no value is returned but control goes back to the point of the call.

Here is a third version of our program. This time we incorporate the loop inside the function and use a return statement to break out of the loop and the function:

```
def ReadAndReportLoop():
    while True:
        text = input( "Enter a string: " )
        if text == "":
            return
        else:
            print( "You typed '%s'." % text )

def main():
    ReadAndReportLoop()

main()
```

Program 4.2.3: Using return to break out of a function

Since function ReadAndReportLoop() does not return any values, we call it as a statement, not as an expression. Note that the return statement exits from the function even when it is nested inside a loop.

As a final example, here is a function that simulates the toss of a coin. It gets a random value that is either 0 or 1, and returns a corresponding value "Heads" or "Tails". This makes use of the randint() function that we introduced in Section 2.4. This function requires us to include the line:

```
from random import *
```

at the top of the program. Recall that randint(a, b) returns a random integer between a and b, so randint(0, 1) returns either a 0 or a 1. Of course, we could replace a call to function CoinToss() with one to randint(0, 1) but that would lose one of the great advantages of writing our own functions: abstraction. When we see the name CoinToss we know that the function represents the tossing of a coin. This is very useful information for a human trying to understand the program, information that is missing from the expression randint(0, 1).

```
from random import *

def Toss():
    x = randint(0, 1)
    if x == 0:
        return "Heads"
    else:
        return "Tails"

def main():
    for i in range(10):
        print( Toss() )

main()
```

Program 4.2.4: Simulating coin tosses

A function call in Python always gives a value, which is None (a formal value in Python) if the function itself does not return anything. You should design functions so that they either return a value, and in that case they should always return something for any input, or else to not return a value in any situation. The following program will print `"small"` if the user enters a number less than 10, and it will print the word `"None"` if the user enters a value 10 or larger. This is very confusing for the user of the program. The problem comes from function BadSize(), which only returns a value in some cases. Design your programs so this doesn't happen.

```
def BadSize(x):
    if x < 10:
        return "small"

def main():
    value = eval(input("Enter a number: "))
    print( BadSize(value) )

main()
```

Function BadSize sometimes returns None

The problem comes from function BadSize(), which only returns a value in some cases. Design your programs so this doesn't happen.

Scoping Rules

People have been writing programs for over 50 years and for most of that time functions have been the main building blocks of programs. One thing we have learned about the art of programming is that code is much more likely to be correct if the functions are as small as possible and independent of each other. It might sound convenient to allow one function to modify the variables of another, but that turns out to be a very bad idea one that is the cause of many, many mysterious program failures. Python does not allow this to happen. Unless you explicitly state that a variable is **global** (something we will discuss later), all of the variables of a function are hidden from all other functions they are accessible only within the function in which they are created.

Consider the following simple program.

```
def Changer():
    x = 23

def main():
    x = 5
    Changer()
    print( x )

main()
```

At first glance it might seem that function Changer() will modify the value of x to 23 and so this program might print 23. That is not correct the program actually prints 5. The reason for this is that function Changer() and function main() each have their own variables named x. Neither function can see the other function's variables. So main() changes its x to 5 and Changer() sets its to 23; these variables are completely unrelated. The **print**-statement occurs in main(), so it is main's x that is printed, with the value of 5.

If you want a function to change the value of a variable in another function you should have the first function return the new value. Here is this program rewritten correctly.

```
def Changer():
    return 23

def main():
    x = 5
    x = Changer()
    print( x )

main()
```

Program 4.2.5: Changing variables with a function call

Now function Changer() returns the new value to main(), which assigns this value to x. Only function main() can see x, so only main() can change its value. Program 4.2.5 prints 23.

Similarly, the following program gives an error message saying that variable x in the line **if** x > 9: of function Printer() is not defined.

```
def Printer():
    if x > 9:
        print( "big" )
    else:
        print( "small" )

def main():
    x = 5
    Printer()

main()
```

No variable x is created for function Printer(), and this function can't see the variable x inside main().

The correct way to get data into a function is to pass it in through a parameter. That is the topic of our next section.