Microsoft

Begin to Code with Python

Rob Miles

Microsoft

Begin to Code with Python

Rob Miles

Give information to functions using parameters

The greeter function shows how functions can be used, but it isn't really that useful because it does the same thing each time it's called. To make a function truly useful, we need to give the function some data with which to work. You've already seen many functions that are used in this way. The print function accepts items to print. The sleep function accepts the length of time that the program should sleep. We can make a times-table function that accepts the times-table to produce.

times_value parameter

```
def print_times_table(times_value):
    count = 1
    while count < 13:
        result = count * times_value
        print(count, 'times', times_value, 'equals', result)
        count = count + 1
```

Use the value of times_value in the function

A program can use the print_times_table function any time it wants to print a times table. The function accepts a single argument, which is the times table to be produced.

```
print_times_table(5)
```

The statement above would call the print_times_table function and ask it to print out the times table for 5. If we want to see the times table for 99, we just need to change the number that we pass to the function.

```
print_times_table(99)
```

Arguments and parameters

From the title of this section, you might expect that we will have a difference of opinion, but in Python, the word *argument* has a particular meaning. In Python, the word argument means "that thing you give to the call of a function."

```
print_times_table(7)
```

In the above statement, the argument is the value 7. So, when you hear the word argument you should think of the code that is making a call of the function.

In Python, the word *parameter* means "the name within the function that represents the argument." The parameters in a function are specified in the function definition.

```
def print_times_table(times_value):
```

This is the definition of the `print_times_table` function. It specifies that the function has a single parameter, which is the name `times_value`. When the function is called, the value of the `times_value` parameter is set to whatever has been given as an argument to the function call. Statements within the function can use the parameter in the same way as they could use a variable with that name.

CODE ANALYSIS

Arguments and parameters

We can find out more about arguments and parameters by looking at the code that uses them.

Question: What would the following program do?

```
# EG7-02 Times Table
def print_times_table(times_value):
    count = 1
    while count < 13:
        result = count * times_value
        print(count, 'times', times_value, 'equals', result)
        count = count + 1

print_times_table(6)
```

Answer: The program prints out the times table for 6.

Question: What would happen if we changed the call of the `print_times_table` function to the one below that has a string as the argument? Would the program fail?

```
print_times_table('six')
```

Answer: The program doesn't fail, but it does something you might not expect.

```
1 times six equals six
2 times six equals sixsix
3 times six equals sixsixsix
4 times six equals sixsixsixsix
5 times six equals sixsixsixsixsix
6 times six equals sixsixsixsixsixsix
7 times six equals sixsixsixsixsixsixsix
8 times six equals sixsixsixsixsixsixsixsix
9 times six equals sixsixsixsixsixsixsixsixsix
10 times six equals sixsixsixsixsixsixsixsixsixsix
11 times six equals sixsixsixsixsixsixsixsixsixsixsix
12 times six equals sixsixsixsixsixsixsixsixsixsixsixsix
```

It turns out that Python is able to perform the multiplication operation between strings and numbers.

The statement below is the one in print_times_table that works out the result. It takes the count (which goes from 1 to 12) and multiplies it by the times_value (which is a parameter in the function).

```
result = count * times_value
```

Multiplying two numbers will produce a numeric result. Multiplying a string by a value will repeat the string the number of times equal to the product. This illustrates an important principle of the Python language. It will decide what to do based on the type of things with which it is working. This can lead to programs that don't do what you might expect.

Question: How do we make the print_times_table function work with integer parameters only?

Answer: Before we decide to fix this problem, we must decide whether we need to fix it at all. If we're using this function in a program that's already checking the input values, then perhaps we don't have to worry about this issue.

If we do try to fix the problem, we must know what should happen. Should the function print a warning message? Should it stop the program? Deciding on an error strategy is an important part of program design, and you should do this in consultation with the customer (if you have one).

In this case, we might decide to be very strict and make the print_times_table function cause an error if it is not given an integer to work with. It turns out that Python has a built-in function called isinstance that a program can use to check whether a given item holds a particular type of data. The isinstance function accepts two arguments, the item to be tested and the type we are checking. It returns True if the item is of the given type, and False if not.

```
# EG7-03 Safe Times Table
if isinstance(times_value,int)==False:
    raise Exception('print_times_table only works with integers')
```

Test the type of the times_value

Raise an exception if the type is not integer

The statements above show how we could use `isinstance` to cause an exception to be raised if the parameter to the function is invalid. The first statement performs the test to see if the function has been given an integer. The second statement is one we haven't seen before. The second statement raises an exception, which causes the program to stop with an error.

```
Traceback (most recent call last):
  File "C:/EG7-03 Safer Times Table.py", line 11, in <module>
    print_times_table('six')
  File "C:/ EG7-03 Safer Times Table.py", line 4, in print_times_table
    raise Exception('print_times_table only works with integers')
Exception: print_times_table only works with integers
```

You can think of an `Exception` as a chunk of data that describes why something went wrong. When an exception is created, it is given a string that describes the error. The exception can be picked up in a `try` construction to allow a program to deal with errors, as we saw in the section "Exceptions and number reading" in Chapter 6. We'll cover exceptions in detail later in the text.

Multiple parameters in a function

A function can have multiple parameters. Currently, the `print_times_table` function always prints out 12 results, starting with 1 times the `times_value` and ending with 12 times the `times_value`. If we are printing out tables for mathematical geniuses, we might want to produce a times table that goes up to 20 times the input value. Alternatively, some people might prefer smaller tables that only go up as far as five times. We could write a different function for each of these table sizes, or we could make the function more flexible by making it accept the size of the times table as well as the number to multiply.

```
# EG7-04 Two Parameter Times Table
def print_times_table(times_value, limit):
    count = 1
    while count < limit+1:
        result = times_value * count
        print(count, 'times', times_value, 'equals', result)
        count = count + 1
```

This version of the function has two parameters. The first parameter, `times_value`, is the number for which times table is desired; the second parameter is the `limit` for the table to be produced.

Now let's call the function.

```
print_times_table(6, 5)
```

The statement above would call the `print_times_table` function and ask for the times table for 6 up to 5 times 6.

```
1 times 6 equals 6
2 times 6 equals 12
3 times 6 equals 18
4 times 6 equals 24
5 times 6 equals 30
```

Positional and keyword arguments

Consider the following function call.

```
print_times_table(12, 7)
```

The statement above makes a call of the `print_times_table` function, but you might be forgiven for wondering whether it prints out the times table for 12 or the times table for 7. You might need to go back to the original code to check the sequence in which the arguments (12 and 7) are mapped to the parameters (`times_value` and `limit`). Arguments mapped in this way are called *positional* arguments because the positions of the arguments given to the function and the parameters defined in the function determine which argument value maps to which parameter. In other words, the sample above would print the times table for 12, because the `times_value` parameter was given first in the original definition.

To make things easier for programmers, Python allows you to use keywords to identify the arguments to a function when you call it.

```
# EG7-05 Keyword Arguments
print_times_table(times_value=12, limit=7)
```

If you use keyword arguments, you don't have to worry about getting the order of the arguments correct when you call functions.

```
print_times_table(limit=7, times_value=12)
```

This call of the print_times_table function will produce the same result as the previous one. I find keyword arguments very helpful. When I write a Python function that accepts more than one argument, I try hard to use keyword arguments for every call of that function.

WHAT COULD GO WRONG

Don't mix positional and keyword arguments

Python will let you mix positional arguments and keyword arguments in a call to a function. However, it can be hard to work out what is going on when you do this. I strongly suggest using either all positional arguments (if it is obvious what the arguments mean) or all keyword arguments.

Default parameter values

When we created the first print_times_table function, we assumed that the limit of the times table to be produced was 12. In other words, the output would go from "1 times" up to "12 times." Then we allowed the user to specify the limit. However, most users of our function will want to go up to a limit of 12 times. We can reflect this by providing a default value for the limit parameter.

Default value for the limit parameter

```
# EG7-06 Default parameters
def print_times_table(times_value, limit=12):
    count = 1
    while count < limit+1:
        result = times_value * count
        print(count,'times', times_value, 'equals', result)
        count = count + 1
```