Writing your own functions

Well done for making it to the final week of the challenge!

This week we have got some really interesting questions lined up for you.

You have been calling functions for several weeks now, either builtin functions like `raw_input`, functions from modules like `random.shuffle`, or methods like `reverse` for lists.

Now that your programs are becoming more complicated it is time for us to show you *how to write your own functions*. This will allow you to make your programs simpler, and therefore easier to understand and debug.

1 Functions

A *function* is an independent, named chunk of code that performs a specific operation. It can be run or *called* by referring to it by name with any information called *arguments* it needs to do its job. By now you will have had lots of experience calling builtin functions like `raw_input` or `int`. Both `raw_input` and `int` take a single argument. For `raw_input` this is the message to display to the user, e.g. `'Enter a number: '`, and for `int` it is the value to be converted into an integer, e.g. `'10'`.

Different languages (and textbooks) sometimes have different terms for functions. They may also be called *routines*, *subroutines*, *subprograms*, *procedures*, *messages* or *methods*. Many languages differentiate *procedures* from *functions* by saying that *functions* return a value but *procedures* don't. Since languages like C/C++ and Java have a `void` or empty return type (equivalent to `None` in Python) the distinction no longer makes sense.

Using functions in Python is easy — so easy that we have been using them without really bothering to properly introduce function calls. On the whole, defining your own functions in Python is straightforward. However, there are a number of tricks that can catch you out occasionally. We mention some of those things below.

The clever thing about functions is that the same bit of code can be called again and again, rather than having to write it out multiple times. If we need to make a change to that code we only have to do it in one place rather than each copy. Also, we can use functions to hide complicated code so we don't have to think about it so carefully when we use it.

2 Function Calls

You have already an old hand at calling functions, but let's recap quickly to make sure you understand it all. To call a function you need to how many arguments the function is expecting and what type of values they are. For instance, the `abs` builtin function takes one argument which must be an integer, float (or complex number). Hopefully, if the function's purpose is well defined it will be obvious what information the function needs and why. That is, unless you have a number, then it is meaningless to ask about its absolute value!

The examples below show what happens when you give `abs` the wrong number or type of arguments:


```
>>> abs(1, 3)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in -toplevel-
    abs(1, 3)
TypeError: abs() takes exactly one argument (2 given)
>>> abs()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in -toplevel-
    abs()
TypeError: abs() takes exactly one argument (0 given)
>>> abs('a')
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in -toplevel-
    abs('a')
TypeError: bad operand type for abs()
```

A function call consists of the function name, which may need to be preceded (or *qualified* by the module name, followed by the arguments surrounded by parentheses. A common mistake is to forget to put the brackets when the function takes no arguments.

```
>>> abs
<built-in function abs>
```

Unfortunately, **abs** without the parentheses is not an error since **abs** is a variable which holds the actual code that performs the absolute value. Typing **abs**, is asking to see the contents of the variable, not call the function.

3 Creating your own functions

Functions in Python are created using a *definition* statement. The **def** creates a function from the indented *function body* and assigns it to the name of the function.

```
>>> def hello(name):
...   print 'Hello', name
...
>>> hello('James')
Hello James
```

A definition statement consists of the keyword **def**, followed by the function name (here **hello**), and then a list of zero or more arguments in parentheses (here just one, **name**). The parentheses must be present even when the function takes no arguments. The **def** line must end with a colon (like control statements). The code block of the function must be indented as in control statements.

The **def** statement is executed in the same order as other statements which means your program must reach the **def** statement before it tries to call the function:

```
hello('James')

def hello(name):
  print 'Hello', name
```

When you run this, Python spits out the following error:

```
Traceback (most recent call last):
  File ``example.py'', line 3, in ?
    hello('James')
NameError: name 'hello' is not defined
```

This is because Python attempts to run the call to **hello** before the function itself actually exists, which is why the name **hello** is undefined.

Here is a function that performs *factorial*, that is, for a number n it calculates $n \times n$ - $1 \times ... \times 1$, so 5 factorial, written *5!*, is $5 \times 4 \times 3 \times 2 \times 1 = 120$. The Python version is:

```
>>> def fact(n):
...   result = 1
...   while n > 1:
...     result *= n
...     n = n - 1
...   return result
...
>>>
```

Running this code with 3 calls to **fact** gives:

```
>>> print fact(5), fact(10), fact(20)
120 3628800 2432902008176640000
```

As you can see those factorial numbers get very large very quickly!

4 return Statements

As we have seen, some functions (such as **fact** above) need to give the result of their execution back to the caller. This is done using a return statement.

```
>>> def add(x, y):
...   return x + y
...
>>> r = add(5, 10)
>>> r
15
```

A return statement *immediately* terminates the running function and control is passed back to the caller, that is, the line of code that called the function continues. If return is given an argument, then it is *returned* to the caller and can be stored in a variable, used in an expression, or discarded.

If return is not given a value, then no value is returned to the caller (None is returned instead). This is the same as a function which has finished executing and does not have a return statement. The **hello** function above is an example without a return. Because a return statement terminates the execution immediately, it is often used to exit from a function before the last line of code.

```
>>> def hi():
...   print 'hello'
...   return
...   print 'there'
...
>>> hi()
hello
```

In this example, print 'there' will never get called because the return statement, will cause the function to exit before the second print is reached.

As in the above examples, when a function needs to return an argument, the return statement is followed by a single expression which is evaluated and returned to the caller. A return statement can appear anywhere, including inside if, for and while statements, in which case the function immediately returns.

```
>>> def day(name):
...   if name in ['Saturday', 'Sunday']:
...     return 'weekend'
...   elif name in ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']:
...     return 'weekday'
...   else:
...     return 'not a day'
>>> day('Monday')
'weekday'
>>> day('Sunday')
'weekend'
```

In this function there are three separate `return` statements that correspond to the three different conditions that this function handles.

5 Arguments

Communicating information to a function can occur in two ways: Either the function declares a number of *arguments* (or *parameters*) which must be passed to it when the function is called, or the function uses variables that are defined in the main program.

You have probably guessed from the examples above how arguments can be used because it's kind of hard to do interesting examples without them. That's not surprising since arguments provide the information functions need to do their work.

Adding arguments to a function is as simple as adding variable names between the parentheses in the `def` statement. Then when the function is called, the value you want that argument to have in that call will need to be added between the parentheses too. The names of the variables inside (the function definition) and outside (in the function call) need not be the same. For this week's problems it is relatively easy since our checking code will tell you if you have defined your functions incorrectly.

Arguments are the trickiest part of creating functions because they don't quite behave how you might expect. The biggest surprise is that arguments (and variables) that are declared as part of a function are not visible outside that function (in the main program or other functions):

```
>>> def hello(name):
...   print 'Hello', name
...
>>> hello('James')
Hello James
>>> name
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in -toplevel-
NameError: name 'name' is not defined
```

Another complication is that you can't change values that are passed into a function:

```
>>> def addone(x):
...   x += 1
...
```

You might expect that if we call this function on a variable, say **y**, then the value of **y** would be incremented by one after the function call:

```
>>> y = 5
>>> addone(y)
>>> y
5
>>>
```

Outside the function nothing has changed, but **x** inside the function would have been incremented by one. This is because Python functions make a copy of their arguments (called **pass by value**) rather than access the original variables.

So in the case above, a new local variable **x** has been creating containing a copy of the contents of the variable **y** that was passed in as an argument during the function call. When we modify **x** inside the function we are only modifying that local variable.

What this means is that if we want to modify a variable we need to pass it in to the function and then return it, like this:

```
>>> def addone(x):
...   return x + 1
...
>>> y = 5
>>> y = addone(5)
>>> y
6
>>>
```

There is one final complication with passing by value, and that is that Python data structures are stored as references. We don't want to go into the details of what that means now, but the gotcha is that they can be modified (but not assigned) inside a function call:

```
>>> def appendone(x):
...   x.append(1)
...
>>> y = []
>>> appendone(y)
>>> y
[1]
>>> appendone(y)
>>> y
[1, 1]
>>>
```

6 Testing functions in IDLE

In the examples above we have developed our functions in the Python shell rather than in a separate file IDLE. However, you normally develop programs in separate Python files and then run them. So how do you do that for your own functions?

Say you have just written your own function in a separate file in IDLE and you want to test it. For example:

```
def greet(name):
    return 'Hello ' + name
```

You can test this in the interactive interpreter in the same way by pressing F5 (or selecting **Run Module** from the **Run** menu). But when you do this nothing seems to happen because you have no code apart from the `def` statement in your file, and the `def` statement doesn't produce any output on the Python shell:

```
>>> ================================ RESTART ================================
>>>
```

What you need to do then is call your function manually:

```
>>> greet('James')
'Hello James'
>>>
```

7 More dictionary methods

Sometimes you'll need to either retrieve the value for a particular key from the dictionary, or use a default value if the key doesn't exist. This typically takes 3 lines of Python:

```
>>> val = 'default'
>>> if key in d:
...     val = d[key]
>>>
```

However, getting a key-value pair or default is so common, dictionaries provide the **get** method to simplify this:

```
>>> val = d.get(key, 'default')
```

The first argument to **get** is the key. If the key is in the dictionary then the corresponding value is returned, otherwise the second argument to **get** (here **'default'**) is returned.

If you want to copy the key-value pairs from one dictionary to another, Python provides the update method:

```
>>> d1 = {'a': 1, 'b': 5}
>>> d2 = {'a': 4, 'c': 6}
>>> d1.update(d2)
>>> d1
{'a': 4, 'c': 6, 'b': 5}
```

Notice that not only does the new key **'c'** have the value **5** from **d2**, but the value for **'a'** has also been updated to the value from **d2**.

8 Miscellaneous string tricks

To make a couple of the questions a bit easier, here are a last couple of miscellaneous tricks:

Firstly, the string module has some useful constants:

```
>>> import string
>>> string.lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?%%[\\]^_`{|}~'
>>>
```

If you need to strip off all punctuation, you can now use:

```
>>> x = "'Hello!'"
>>> x.strip(string.punctuation)
'Hello'
>>>
```