Chapter 3

Functions

3.1 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')
<type 'str'>
```

The name of the function is `type`. The expression in parentheses is called the **argument** of the function. The result, for this function, is the type of the argument, which is a string.

It is common to say that a function "takes" an argument and "returns" a result. The result is called the **return value**.

When you call a function in interactive mode, the interpreter displays the return value, but in a script a function call, all by itself, doesn't display anything. To see the result, you have to print it:

```
print type('32')
```

Or assign the return value to a variable, which you can print (or use for some other purpose) later.

```
stereo = type('32')
print stereo
```

3.2 Type conversion functions

Python provides built-in functions that convert values from one type to another. 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(): 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
```

`long` can accept any numerical type and even strings of digits:

```
>>> long(1)
1L
>>> long(3.1415)
3L
>>> long('42')
42L
```

Finally, `str` converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

3.3 Math functions

Python has a math module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions.

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' from '/usr/lib/python2.4/lib-dynload/mathmodule.so'>
```

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)
0.707106781187
```

The expression `math.pi` gets the variable `pi` from the math module. Conveniently, 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
0.707106781187
```

3.4 Composition

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

And even function calls:

```
x = math.exp(math.log(x+1))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. An expression on the left side is a syntax error.

```
>>> minutes = hours * 60                # right
>>> hours * 60 = minutes                # wrong!
SyntaxError: can't assign to operator
```

3.5 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 (the one on disk).

Debugging can take a long time if you keep running the same, incorrect, program over and over! And that brings me to the Third Theorem of Debugging:

> 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!

3.6 Glossary

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.

module: A file that contains a collection of related functions and other definitions.

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 values defined in a module.

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.

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

3.7 Exercises