Chapter 8

Functions

“In the old days when people invented a new function they had something useful in mind.”
— Henri Poincare

“It is the function of vice to keep virtue within reasonable bounds.”
— Samuel Butler

“The smaller the function, the greater the management.”
— C. Northcote Parkinson

Programming languages are connected to an area of computing called software engineering, which asks a simple question: why is the production of software different from designing bridges, manufacturing aircraft, developing pharmaceuticals, or building houses? Architects and construction firms are able to pretty well predict how long it will take to build a house and how much it will cost, but the same cannot be done with software. Also, when it comes to building a house, there is an established marketplace of parts, concrete, structural beams, and so on; also there are tools for all aspects of construction. The dream of software engineering is to find the right kinds of tools and “parts” so that creating programs and computing applications can become predictable, reliable, and have acceptably low cost. This dream has influence the evolution of programming languages and industrial conventions for organizing software systems.

One important discovery of software engineering is the DRY principle (“Don’t Repeat Yourself”). The basic idea is simple: humans should not need to write software that is repetitive — instead, we should have tools so that a particular algorithm or symbolic manipulation is done once, and then reused, instead of rewriting the same thing over and over. A key technical device for this purpose is the invention of the subroutine. A subroutine is a portion of a program that can be reused, possibly for multiple reasons; the goal of a subroutine is typically some limited computation that returns a value based on information given to the subroutine. Suppose a subroutine exists for calculating $\sqrt{x}$, call
it \texttt{sqrt}(x). Within a program, there can be expressions like \texttt{sqrt(801.25)}, \texttt{sqrt(4.1129e12)}, or \texttt{sqrt(169)}. The method for calculating a square root is a highly researched and optimized numerical procedure, and instead of doing the work of figuring out how to do this ourselves and repeating the effort, it’s handy to have an “off-the-self” subroutine available.

**Function Syntax**

Python uses the term *function* rather than subroutine, which can be misleading considering all the different meanings of “function” outside of computing. You can define as many functions as you like in Python, even at an interactive session:

```python
>>> def sqrt(x):
...     return x**0.5
... 
... sqrt(16)
4.0
>>> sqrt(900)
30.0
>>> sqrt(2)
1.4142135623730951
```

The syntax starts with the *keyword* \texttt{def}, then a name of a function, a *parameter list*, and a colon. Thereafter follow one or more lines, one of which is a \texttt{return} statement. The return statement provides an expression that Python evaluates, reducing it to a value that will be substituted for a function application. For instance, the function application \texttt{sqrt(16)} first *binds* the value 16 to the symbolic name \texttt{x}, then evaluates the expression \texttt{16**0.5}, which turns into 4.0 by reducing the expression. The “binding” of 16 to the symbolic name \texttt{x} is temporary, done only for the duration of Python’s evaluation of function application; later, values 900 and 2 will be bound to the symbolic name \texttt{x} when they are used in \texttt{sqrt} application.

Notice the \ldots shown by Python during the definition of the function \texttt{sqrt}; this was done by Python because it is aware that a function is being defined, and expects one or more lines of code to be part of the function. *It is crucial that these lines be indented*: you will see many examples illustrating this point. Python uses the notational convention of indenting lines of text to tell where a definition starts and where it ends. In the interactive session, a blank line ends the definition.

Rather than define functions interactively, we can put the functions into a script. Suppose \texttt{myscript.py} is a file containing these lines (using Python3 syntax for the print statements).
```python
def firla(s):
    return s[0]+s[-1]
print( firla("hold") )
print(firla('taxes'))
print(firla("Garden Ornament"))
```

Running the script has this output:

```plaintext
> python myscript.py
hd
ts
Gt
```

The function `firla` returns a string consisting of the concatenation of the first and last character of the function argument. One way to interpret the function definition, which appears first in the script file, is that it's a command: the command is to define a function. Once the function is defined, it can be freely used in expressions later in the script.

A script can contain many function definitions, each following the syntax of function definition.

```python
def firla(s):
    return s[0]+s[-1]
def lafir(s):
    return s[-1]+s[0]
print(firla("hold"))
print(lafir("hold"))
```

**Terminology**

Looking further into the syntax of function definition, function application, and related conventions, it's helpful to develop some terminology. Much of this terminology is used in other programming languages as well, and is generally “common knowledge” among programmers.

**Head and Body** Some jargon for function definition is head and body of the function. The “head” is just the first line, starting with `def` and ending with a colon. The remaining lines of the function definition are the body of the function, which have to be indented so that Python knows which lines belong to the function body.
Parameters vs Arguments  Perhaps you’ve noticed that sometimes functions are described as having parameters, and at other times they are arguments? Are these terms just synonyms? To many students, there seems no difference. Technically there is a clear difference in usage.

On the defining line of a function (beginning with the keyword `def` and ending with the colon), the symbolic names in the function header are called parameters. Subsequently, for each time that function is used in application, what gets evaluated and substituted for the parameter is called an argument to the function. A program has one line for defining parameters, but can have any number of different arguments to the function, because there may be many function applications.

```
def sqrt(x):
    x is a parameter
...
print sqrt(100) 100 is an argument
print sqrt(225) 225 is an argument
```

Binding  When a Python program evaluates a function application, function arguments get bound to the parameters defined for that function. Later in the chapter we will see several ways of binding arguments to parameters. One important observation about binding is that if an expression is used as an argument, Python must first evaluate that expression, reducing it to a value; only then does the symbolic name used as the function parameter have a definite meaning inside the function’s body.

Command Functions  Functions can return values based on expressions of their arguments, but Python does not require all functions to return values. Some functions are like Python3’s `print` syntax, essentially using the function syntax to accomplish some external effect: drawing on a window, playing a sound, etc. A simple example, using Python3 syntax is this script:

```
def twiceprint(sometext):
    print(sometext)
    print(sometext)
twiceprint("Hello")
twiceprint("there")
```

The function `twiceprint` has no return statement, since the point is to print the argument twice, but there is nothing to calculate. Similarly, each function evaluation is simply a line with `twiceprint` and an argument, and nothing is expected to be computed. When run, the output of this script is:

```
Hello
Hello
```
The same script written using Python2 syntax is:

```python
def twiceprint(sometext):
    print sometext
    print sometext
twiceprint("Hello")
twiceprint("there")
```

Function application is the same for Python2 and Python3; the only difference is the `print` statement. Typical command functions may not use any parameter at all: in such cases the parameter list is empty:

```python
def shoutout():
    print "one"
    print "more"
    print "time"
shoutout()
shoutout()
shoutout()
```

Notice how the indentation of all the lines in the body is the same. Python does not have a precise requirement of how much the lines in a function body need to be indented, but they **must** all be indented by the same amount: all the lines in the body should line up on the left.

**Tracing** The ability to insert `print` commands into a function body can help us see how Python works, or perhaps that a function has been incorrectly coded. Consider this definition of a function `calc` and two function applications, in a script:

```python
def calc(y):
    print "argument is", y
    return y**(1.0/3.0)/(1.0/y)
print "A:", calc(2**3)
print "B:", calc(2**5)
```

When this runs, the output is:

```
A: argument is 8
 16.0
B: argument is 32
101.593667326
```
What’s surprising about this is that the two `print` command mingle their output. The first `print` statement to run outputs `A:`, but then must wait for the function application `calc(2**3)` to be evaluated and reduced to a value before it outputs the rest. However, inside of the function application `calc(2**3)` there is a `print` statement (by the way, this adds a newline, as `print` commands normally do). Finally, the value 16.0 is returned, and then output.

**Parameter Specification**

To this point, the only functions introduced either used a single parameter or had an empty parameter list. Technically this is already enough for all programming needs, because a single parameter could be a tuple, dictionary, or list type – each of which can bundle up any collection of data. However, it’s usually more convenient to use a parameter list with several symbolic names, one for each parameter.

**Positional Parameters**  We start with a simple example.

```python
def subtract(a,b):
  return a-b
```

Evaluation of `subtract(8,2)` returns 6, whereas `subtract(2,8)` returns -6. Thus, the first argument binds to `a` and the second argument binds to `b`. This style of matching up arguments, first argument to first parameter and so on, is called the **positional parameter** style. Here’s a three parameter example:

```python
def subthenadd(a,b,c):
  return a-b+c
```

When the head of the function contains, say three parameters, any function application needs to have three parameters. So, Python will output an error when the wrong number of arguments is tried for function application:

```python
>>> subthenadd(9,2)
TypeError: subthenadd() takes exactly 3 arguments (2 given)
```

Python has some advanced notation to allow the definition of functions with a variable number of parameters (for instance, like Python3’s `print` function, which can have any number of arguments).
Function Type The error in the previous example was reported by Python as a `TypeError`. This error suggests that functions have types. You can see this by asking Python with the `type()` query:

```python
>>> type(subthenadd)
<type 'function'>
```

Python is thus aware of the function name much like a data type, such as string, list, or numeric types. Further, the number of parameters is considered to be a characteristic of the type of function, as are some other features described below.

Keyword Arguments Python differs from traditional languages, which use the positional style only for functions and subroutines, by offering a variety of styles for application and definition of parameters. First, we show how even functions defined with positional parameters can be called with keyword style arguments. This style uses the “=” symbol to say which argument should be bound with a parameter. The first example refers to the `subtract` function defined above.

```python
>>> subtract(a=9,b=2)
7
>>> subtract(b=1,a=3)
2
>>> subtract(b=100)
TypeError: subtract() takes exactly 2 non-keyword arguments (1 given)
>>> subtract(16,b=8)
8
>>> subtract(10,a=5,b=0)
TypeError: subtract() got multiple values for keyword argument 'a'
```

The examples shows several features of calling a function with the keyword argument style:

- The arguments bind by the parameter name, `a` and `b` in the example. It doesn’t matter which is the order of the keyword arguments given.
- All parameters defined in the function header need to be bound by the function application.
- An argument for a parameter can only be specified once in the function application.
- Positional and keyword argument styles can be mixed. The rule here is that positional arguments come first, then any remaining parameters can be specified by keyword.

The keyword style of argument specification is not so useful for invoking functions defined with positional parameters; the real purpose of this style is motivated by the next topic, keyword parameters.
Keyword Parameters  Parameters can also be specified as keywords by using “=” and a value to bind to the parameter. During function application, a keyword argument then overrides the parameter.

```python
def nand(left=True, right=True):
    return not (left and right)
...
>>> nand(left=False, right=True):
    True
>>> nand(left=5>3, right='a' in "lost")
    True
```

The parameter for `nand`'s definition says that the symbolic name `left` has the value `True`; however, the first function application contradicts this, instead specifying that `False` should be bound to `left`. When Python evaluates the function application, it's what is given in the argument that takes over. Though keyword style parameters can be used with the argument style of function application, seen above, the positional style can still be used:

```python
>>> nand(False, True)
    True
>>> nand(True, True)
    False
```

The value given for a parameter in the function header is called the default value. There are two motivations for the keyword style: first, people may forget the exact order of what are the parameters to a function, but remember the symbolic names of the parameters. The second motivation is that there may be a parameter which, in 99% of all function applications, tends to get the same value. To streamline programming notation, in the spirit of DRY, Python allows parameters that have default values to be omitted from function application. When omitted from the arguments, the value used in function evaluation will be the default value.

```python
>>> nand(right=(16==4*4))
    False
>>> nand(left=False)
    True
>>> nand()
    False
```

The last line above shows that all arguments can be omitted, because the header for `nand` gives default values for all parameters.

Beginning Python programmers tend to avoid using keyword parameters, except where necessary: when using some of Python's library of functions, the use of keywords is practically the only way to take advantage of some functions.
The **None Type** Functions that produce a result based on arguments or default values of parameters have **return** statements, to say what is the result value. We have seen that command functions (like ones that only print) do not have **return** statements. You should know, however, that Python actually does have a return value even for functions without **return** statements: it is a special value called **None**. There is even a special type in Python for this:

```python
>>> type(None)
<type 'NoneType'>
>>> None
```  
Unlike other values, you see above that Python’s display of **None** in an interactive session shows nothing. But, you can see it as part of a tuple or list:

```python
>>> [None,False]
[None, False]
```  
Here is a demonstration of how **None** gets produced for a function which does not have a **return** statement:

```python
def noret(x):
    x + 1

>>> [noret(5)]
[None]
```  
If you were expecting to see 6 as the result of `noret(5)`, then you’ve missed an important point about Python functions: they return **None unless** you have a **return** statement. (Of course, if you really wanted to, you could have a “**return None**” in your function, but why would anyone do that?)

**Naming**

So far, the chapter has not addressed the concern of how functions and parameters are **named**. Previously, function names `sqrt`, `firla`, `lafir`, `twiceprint`, `shoutout`, `calc`, `subtract`, `subthenadd`, `nand`, `noret` have been defined as examples. These all happen to be lowercase function names, but uppercase names and mixed names are also permitted, for instance

```
MUL, Fraction, CaLiFoRnIa, EasyThereGuy
```
Functions

are valid Python function names. Generally, any combination of alphabetic characters (without any blanks) is an acceptable name. Python discriminates between upper and lowercase names, so `sqrt` and `sqrt` are considered to be different names. The only restriction is that certain reserved names, also called reserved words and reserved identifiers, should not be chosen as function names. Examples of this are: `and`, `or`, `not`, `False`, `True`, `in`, `def`, `type`, `len`, `print`, `None`, and any other part of the Python language that occurs in statements, types, operators, and such – it can be confusing to define functions with names that conflict with the language.

Similarly, parameter names which symbolically refer to arguments can be any name that does not conflict with the Python language; above examples use parameter names `x`, `y`, `a`, `b`, `left`, `right`, but as with function names, you can use upper and lower case or a mixture of cases. As a matter of style, some people prefer to use names that reflect the “meaning” of a parameter and so names like `dividend` or `InvoiceNumber` could be used. Other people like terse names like `x`, `y`, `J`, and so on. Avoid tricky and confusing names like `ILIIIL`, `LLILIIIL`, etc, which make reading code very difficult.

Numeric digits can also be used in names, though not as the first character. Thus names like `y30`, `alpha1`, `Key003` are admissible in Python.

Building on Functions

It should come as no surprise that you can use function application in expressions, even those that are arguments to functions:

```python
>>> nand( subtract(7,3)>0, subtract(2,9)>0 )
True
```

To bind arguments of `nand` to its parameters, Python first has to evaluate the expressions for each argument, which are function applications. Sometimes this is called nested evaluation, where in order to evaluate one thing, something inside (here, it’s inside the argument list) has to be evaluated. This use of “nest” is an analogy to nesting dolls, a Russian folk toy¹. As well as putting function application inside of argument expressions, it is quite common to organize function definitions that use function application in their bodies. For example, the formula for distance between two points in two dimensional space, called the Euclidean distance, is

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Using Python definitions, this formula could be written with several functions:

¹Look up matryoshka if you’re curious.
```python
def dimdiff(a, b, ind):
    return a[ind] - b[ind]
def xdiff(v, w):
    return dimdiff(v, w, 0)
def ydiff(v, w):
    return dimdiff(v, w, 1)
def sqrt(x):
    return x ** 0.5

def distance(p, q):
    return sqrt(xdiff(p, q)**2, ydiff(p, q)**2)
```

This example probably went too far in using functions, but it makes the point that you can write one function in terms of expressions that rely on other functions. Whether this is a good idea or not depends on the situation.

### Built-in Functions

Python has some functions that are automatically available without needing the programmer to define them. These are called or built-in functions. Here is a partial list of Python’s built-in functions:

- `abs(x)`: returns absolute value
- `min(x,y)`: returns smaller argument
- `max(x,y)`: returns larger argument
- `type(x)`: type query
- `len(x)`: length of sequence

Chapter 16 describes how thousands more functions can be brought into Python, thanks to libraries that are freely available with Python or found via the web.

### Type Conversion Functions

Another set of built-in functions are ones that can convert data from one type to another. The function names are the same as the type names, and all take a single argument. Also, all of these have a default value when no argument is given. The following shows the default values:

```python
int() 0
float() 0.0
bool() False
str() ''
tuple() ()
list() []
dict() {}
set() set([])
```
Common examples of type conversion are string to numeric, float to integer, and anything to a string. Here are some examples:

```python
>>> int(1.95)
1
>>> float(999)
999.0
>>> int(False)
0
>>> tuple([False,-9,"abc"])
(False, -9, 'abc')
>>> list("state")
['s', 't', 'a', 't', 'e']
>>> str([1,2,3])
'[1, 2, 3]'
```

Conversion to a string again requires that you comprehend the difference between typed data and the characters that represent that data. This often trips up beginners. Consider these cases:

```python
>>> [8,2,5,1][2]
5
>>> str([8,2,5,1])[2]
',
>>> "home" + 9
TypeError: cannot concatenate 'str' and 'int' objects
>>> "home" + str(9)
'home9'
>>> int("seven")
ValueError: invalid literal for int() with base 10: 'seven'
>>> int("7")
7
```

The examples aren’t a complete set of rules for how Python does (and does not) convert between types, but shows the more common cases. The `print` function in Python3 uses the equivalent of `str()` for each argument before output; the `print` statement in Python2 similarly handles each expression given on the statement.

**Namespace Queries**

During a Python interactive session, or during a the run of a script, Python builds an internal dictionary of the names of defined functions and other things needed to evaluate expressions and to execute programs. There is a built-in
function \texttt{dir} that returns a list of what are the currently defined names. Here is the output of \texttt{dir()} when Python first starts an interactive session:

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

All the special names used by Python start with underscore characters – these have special significance for the Python environment. Now suppose the session continues with:

```
>>> def sqrt(x):
...     return x**0.5
...
>>> def double(x):
...     return 2*x
...
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sqrt', 'double']
```

This example hints at how Python does its job: it uses concepts from Python itself, in order to interact with users, evaluate what they type, define functions, and so forth. It uses integers, strings, lists, dictionaries and other data types, with extensive use of functions to do all the work.

## Composition

Building new functions in terms of old ones is a standard way to write software. A term for this is \textit{function composition}, which means putting functions together in some combination to get new functions. For some examples, recall that Chapter 5 explained how the operators \texttt{==} and \texttt{<} work for sequences, but did not define other comparisons between sequences. Using function composition it is easy to explain how the operators work. Below, let \texttt{leq} be a function for the \texttt{<=} operator, \texttt{neq} for \texttt{! =} (and its synonym \texttt{<>}), \texttt{gt} for \texttt{>}, and \texttt{geq} for \texttt{>=}.

```
def geq(x,y):
    return not (x < y)
def neq(x,y):
    return not (x == y)
def gt(x,y):
    return geq(x,y) and neq(x,y)
def leq(x,y):
    return not gt(x,y)
```

Function composition has another meaning for a style of programming called \textit{functional programming}. The idea of functional programming is that parameters of functions, and more broadly inputs to algorithms can be \texttt{functions} as well as ordinary data values. It’s hard to appreciate this without looking at an example.
def addOne(x):
    return x+1

def addTwo(x):
    return x+2

def appFtoVal(f,v):
    return f(v)

def appFFtoVal(f,v):
    return f(f(v))
...

>>> appFtoVal(addOne,100)
101
>>> appFtoVal(addOne,100)
102
>>> appFtoVal(addTwo,100)
102
>>> appFtoVal(addTwo,100)
104

What’s interesting here is that you cannot see, looking at the body of `appFtoVal`, just what exactly the function f does. Indeed, Python gives the freedom of selecting what function f will be up to the caller of `appFtoVal`, as the example shows.

**Terminology Review**

Jargon introduced in this chapter includes: the DRY principle, subroutine, parameter, argument, indented lines, head, body, binding, bound to, positional parameters, keyword parameters, default values, keyword arguments, the `None` type, type conversion functions, name space, function composition, functional programming.

**Exercises**

There can be many answers to some of the exercises, but a correct answer will only use material covered in this and earlier chapters. So, if you have an answer that works, but uses more advanced techniques, even if more efficient, that answer does not reinforce what this chapter covers, and so is incorrect here.

(1) Write a function named `allz` that takes a sequence argument, and returns a string of z characters of the same length as the argument. Here are a few examples of how it should behave:
(2) Every student in an algebra class learns the basic formula for the roots of a quadratic,

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

which finds values of $x$ such that $ax^2 + bx + c = 0$. Write a function `quadroot(a, b, c)` that returns the roots, as a pair, for the given parameters.

(3) Write a function `foo(s)` that returns a boolean, given a string $s$: the result is `True` if the first character of $s$ concatenated with the last character of $s$ also occurs as a string within $s$. Here are some examples:

```plaintext
foo("wanted") → False
foo("drainer") → True
foo("scissors") → True
```