Substantial programs are broken up into functions for better modularity and ease of maintenance. Python makes it easy to define functions but also incorporates a surprising number of features from functional programming languages. This chapter describes functions, scoping rules, closures, decorators, generators, coroutines, and other functional programming features. In addition, list comprehensions and generator expressions are described—both of which are powerful tools for declarative-style programming and data processing.

Functions

Functions are defined with the def statement: image The body of a function is simply a sequence of statements that execute when the function is called. You invoke a function by writing the function name followed by a tuple of function arguments, such as a = add(3,4). The order and number of arguments must match those given in the function definition. If a mismatch exists, a TypeError exception is raised.

You can attach default arguments to function parameters by assigning values in the function definition. For example:

```
def split(line,delimiter=','):
    statements
```

When a function defines a parameter with a default value, that parameter and all the parameters that follow are optional. If values are not assigned to all the optional parameters in the function definition, a SyntaxError exception is raised.

Default parameter values are always set to the objects that were supplied as values when the function was defined. Here's an example:

```
a = 10
def foo(x=a):
    return x

a = 5               # Reassign 'a'.
foo()               # returns 10 (default value not changed)
```

A function can accept a variable number of parameters if an asterisk (*) is added to the last parameter name:

```
def fprintf(file, fmt, *args):
    file.write(fmt % args)

# Use fprintf. args gets (42,"hello world", 3.45)
fprintf(out,"%d %s %f", 42, "hello world", 3.45)
```

Function arguments can also be supplied by explicitly naming each parameter and specifying a value. These are known as keyword arguments. Here is an example:

```
def foo(w,x,y,z):
    statements

# Keyword argument invocation
foo(x=3, y=22, w='hello', z=[1,2])
```

With keyword arguments, the order of the parameters doesn't matter. However, unless there are default values, you must explicitly name all of the required function parameters. If you omit any of the required parameters or if the name of a keyword doesn't match any of the parameter names in the function definition, a TypeError exception is raised. Also, since any Python function can be called using the keyword calling style, it is generally a good idea to define functions with descriptive argument names.

Positional arguments and keyword arguments can appear in the same function call, provided that all the positional arguments appear first, values are provided for all non-optional arguments, and no argument value is defined more than once. Here's an example:

```
foo('hello', 3, z=[1,2], y=22)
foo(3, 22, w='hello', z=[1,2])   # TypeError. Multiple values for w
```

If the last argument of a function definition begins with **, all the additional keyword arguments (those that don't match any of the other parameter names) are placed in a dictionary and passed to the function. This can be a useful way to write functions that accept a large number of potentially open-ended configuration options that would be too unwieldy to list as parameters. Here's an example:

```
def make_table(data, **parms):
    # Get configuration parameters from parms (a dict)
    fgcolor = parms.pop("fgcolor","black")
    bgcolor = parms.pop("bgcolor","white")
    width = parms.pop("width",None)
    ...
    # No more options
    if parms:
         raise TypeError("Unsupported configuration options %s" % list(parms))

make_table(items, fgcolor="black", bgcolor="white", border=1,
                  borderstyle="grooved", cellpadding=10,
                  width=400)
```

You can combine extra keyword arguments with variable-length argument lists, as long as the ** parameter appears last:

```
# Accept variable number of positional or keyword arguments
def spam(*args, **kwargs):
    # args is a tuple of positional args
    # kwargs is dictionary of keyword args
```

Keyword arguments can also be passed to another function using the **kwargs syntax:

```
def callfunc(*args, **kwargs):
    func(*args,**kwargs)
```

This use of *args and **kwargs is commonly used to write wrappers and proxies for other functions. For example, the callfunc() accepts any combination of arguments and simply passes them through to func().

Parameter Passing and Return Values

When a function is invoked, the function parameters are simply names that refer to the passed input objects. The underlying semantics of parameter passing doesn't neatly fit into any single style, such as "pass by value" or "pass by reference," that you might know about from other programming languages. For example, if you pass an immutable value, the argument effectively looks like it was passed by value. However, if a mutable object (such as a list or dictionary) is passed to a function where it's then modified, those changes will be reflected in the original object. Here's an example:

```
a = [1, 2, 3, 4, 5]
def square(items):
    for i,x in enumerate(items):
        items[i] = x * x     # Modify items in-place

square(a)      # Changes a to [1, 4, 9, 16, 25]
```

Functions that mutate their input values or change the state of other parts of the program behind the scenes like this are said to have side effects. As a general rule, this is a programming style that is best avoided because such functions can become a source of subtle programming errors as programs grow in size and complexity (for example, it's not obvious from reading a function call if a

function has side effects). Such functions interact poorly with programs involving threads and concurrency because side effects typically need to be protected by locks.

The return statement returns a value from a function. If no value is specified or you omit the return statement, the None object is returned. To return multiple values, place them in a tuple:

```
def factor(a):
    d = 2
    while (d <= (a / 2)):
        if ((a / d) * d == a):
            return ((a / d), d)
        d = d + 1
    return (a, 1)
```

Multiple return values returned in a tuple can be assigned to individual variables:

x, y = factor(1243) # Return values placed in x and y.

or

(x, y) = factor(1243) # Alternate version. Same behavior.

Scoping Rules

Each time a function executes, a new local namespace is created. This namespace represents a local environment that contains the names of the function parameters, as well as the names of variables that are assigned inside the function body. When resolving names, the interpreter first searches the local namespace. If no match exists, it searches the global namespace. The global namespace for a function is always the module in which the function was defined. If the interpreter finds no match in the global namespace, it makes a final check in the built-in namespace. If this fails, a NameError exception is raised.

One peculiarity of namespaces is the manipulation of global variables within a function. For example, consider the following code:

```
a = 42
def foo():
    a = 13
foo()
# a is still 42
```

When this code executes, a returns its value of 42, despite the appearance that we might be modifying the variable a inside the function foo. When variables are assigned inside a function, they're always bound to the function's local namespace; as a result, the variable a in the function body refers to an entirely new object containing the value 13, not the outer variable. To alter this behavior, use the global statement. global simply declares names as belonging to the global namespace, and it's necessary only when global variables will be modified. It can be placed anywhere in a function body and used repeatedly. Here's an example:

```
a = 42
b = 37
def foo():
    global a       # 'a' is in global namespace
    a = 13
    b = 0
foo()
# a is now 13. b is still 37.
```

Python supports nested function definitions. Here's an example:

```
def countdown(start):
   n = start
   def display():            # Nested function definition
       print('T-minus %d' % n)
   while n > 0:
       display()
       n -= 1
```

Variables in nested functions are bound using lexical scoping. That is, names are resolved by first checking the local scope and then all enclosing scopes of outer function definitions from the innermost scope to the outermost scope. If no match is found, the global and built-in namespaces are checked as before.

```
def countdown(start):
   n = start
   def display():
       print('T-minus %d' % n)
   def decrement():
       n -= 1              # Fails in Python 2
   while n > 0:
        display()
        decrement()

def countdown(start):
   n = start
   def display():
       print('T-minus %d' % n)
   def decrement():
       nonlocal n    # Bind to outer n (Python 3 only)
       n -= 1
   while n > 0:
        display()
        decrement()
```

The nonlocal declaration does not bind a name to local variables defined inside arbitrary functions further down on the current call-stack (that is, dynamic scope). So, if you're coming to Python from Perl, nonlocal is not the same as declaring a Perl local variable.

If a local variable is used before it's assigned a value, an UnboundLocalError exception is raised. Here's an example that illustrates one scenario of how this might occur:

```
i = 0
def foo():
    i = i + 1      # Results in UnboundLocalError exception
    print(i)
```

Functions as Objects and Closures

```
# foo.py
def callf(func):
   return func()
```

```
>>> import foo
>>> def helloworld():
...     return 'Hello World'
...
>>> foo.callf(helloworld)     # Pass a function as an argument
'Hello World'
>>>
```

```
# foo.py
x = 42
def callf(func):
   return func()
```

```
>>> import foo
>>> x = 37
>>> def helloworld():
...     return "Hello World. x is %d" % x
...
>>> foo.callf(helloworld)     # Pass a function as an argument
'Hello World. x is 37'
>>>
```

In this example, notice how the function helloworld() uses the value of x that's defined in the same environment as where helloworld() was defined. Thus, even though there is also an x defined in foo.py and that's where helloworld() is actually being called, that value of x is not the one that's used when helloworld() executes.

Closures and nested functions are especially useful if you want to write code based on the concept of lazy or delayed evaluation. Here is another example:

```
from urllib import urlopen
# from urllib.request import urlopen (Python 3)
def page(url):
    def get():
        return urlopen(url).read()
    return get
```

In this example, the page() function doesn't actually carry out any interesting computation. Instead, it merely creates and returns a function get() that will fetch the contents of a web page when it is called. Thus, the computation carried out in get() is actually delayed until some later point in a program when get() is evaluated.

A closure can be a highly efficient way to preserve state across a series of function calls. For example, consider this code that runs a simple counter:

```
def countdown(n):
    def next():
        nonlocal n
        r = n
        n -= 1
        return r
    return next

# Example use
next = countdown(10)
while True:
    v = next()         # Get the next value
    if not v: break
```

In this code, a closure is being used to store the internal counter value n. The inner function next() updates and returns the previous value of this counter variable each time it is called. Programmers not familiar with closures might be inclined to implement similar functionality using a class such as this:

```
class Countdown(object):
    def __init__(self,n):
        self.n = n
    def next(self):
        r = self.n
        self.n -= 1
        return r

# Example use
c = Countdown(10)
while True:
    v = c.next()         # Get the next value
    if not v: break
```

However, if you increase the starting value of the countdown and perform a simple timing benchmark, you will find that that the version using closures runs much faster (almost a 50% speedup when tested on the author's machine).

The fact that closures capture the environment of inner functions also make them useful for applications where you want to wrap existing functions in order to add extra capabilities.

Decorators

A decorator is a function whose primary purpose is to wrap another function or class. The primary purpose of this wrapping is to transparently alter or enhance the behavior of the object being wrapped. Syntactically, decorators are denoted using the special @ symbol.

```
@trace
def square(x):
     return x*x

def square(x):
    return x*x
square = trace(square)

enable_tracing = True
if enable_tracing:
    debug_log = open("debug.log","w")

def trace(func):
    if enable_tracing:
        def callf(*args,**kwargs):
            debug_log.write("Calling %s: %s, %s\n" %
                            (func.__name__, args, kwargs))
            r = func(*args,**kwargs)
            debug_log.write("%s returned %s\n" % (func.__name, r))
            return r
        return callf
    else:
        return func
```

In this code, trace() creates a wrapper function that writes some debugging output and then calls the original function object. Thus, if you call square(), you will see the output of the write() methods in the wrapper. The function callf that is returned from trace() is a closure that serves as a replacement for the original function. A final interesting aspect of the implementation is that the tracing feature itself is only enabled through the use of a global variable enable_tracing as shown. If set to False, the trace() decorator simply returns the original function unmodified. Thus, when tracing is disabled, there is no added performance penalty associated with using the decorator.

When decorators are used, they must appear on their own line immediately prior to a function or class definition. More than one decorator can also be applied.

Generators and yield

If a function uses the yield keyword, it defines an object known as a generator. A generator is a function that produces a sequence of values for use in iteration. Here's an example:

```
def countdown(n):
     print("Counting down from %d" % n)
     while n > 0:
          yield n
          n -= 1
      return
```

If you call this function, you will find that none of its code starts executing. For example:

```
>>> c = countdown(10)
>>>
```

Instead, a generator object is returned. The generator object, in turn, executes the function whenever next() is called (or _ _next_ _() in Python 3). Here's an example:

```
>>> c.next()            # Use c.__next__() in Python 3
Counting down from 10
10
>>> c.next()
9
```

When next() is invoked, the generator function executes statements until it reaches a yield statement. The yield statement produces a result at which point execution of the function stops until next() is invoked again. Execution then resumes with the statement following yield.

You normally don't call next() directly on a generator but use it with the for statement, sum(), or some other operation that consumes a sequence. For example:

```
for n in countdown(10):
     statements
a = sum(countdown(10))
```

A generator function signals completion by returning or raising **StopIteration**, at which point iteration stops. It is never legal for a generator to return a value other than **None** upon completion.

A subtle problem with generators concerns the case where a generator function is only partially consumed. For example, consider this code:

In this example, the for loop aborts by calling break, and the associated generator never runs to full completion. To handle this case, generator objects have a method close() that is used to signal a shutdown. When a generator is no longer used or deleted, close() is called. Normally it is not necessary to call close(), but you can also call it manually as shown here:

Inside the generator function, close() is signaled by a **GeneratorExit** exception occurring on the yield statement. You can optionally catch this exception to perform cleanup actions.

Although it is possible to catch **GeneratorExit**, it is illegal for a generator function to handle the exception and produce another output value using yield. Moreover, if a program is currently iterating on generator, you should not call close() asynchronously on that generator from a separate thread of execution or from a signal handler.

Coroutines and yield Expressions

Inside a function, the yield statement can also be used as an expression that appears on the right side of an assignment operator.

A function that uses yield in this manner is known as a coroutine, and it executes in response to values being sent to it. Its behavior is also very similar to a generator.

```
def receiver():
    print("Ready to receive")
    while True:
          n = (yield)
          print("Got %s" % n)

>>> r = receiver()
>>> r.next()  # Advance to first yield (r.__next__() in Python 3)
Ready to receive
>>> r.send(1)
Got 1
>>> r.send(2)
Got 2
>>> r.send("Hello")
Got Hello
>>>
```

List Comprehension

The general syntax for a list comprehension is:

```
[expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN ]

a = [-3,5,2,-10,7,8]
b = 'abc'

c = [2*s for s in a]           # c = [-6,10,4,-20,14,16]
d = [s for s in a if s >= 0]   # d = [5,2,7,8]
e = [(x,y) for x in a          # e = [(5,'a'),(5,'b'),(5,'c'),
           for y in b          #      (2,'a'),(2,'b'),(2,'c'),
           if x > 0 ]          #      (7,'a'),(7,'b'),(7,'c'),
                               #      (8,'a'),(8,'b'),(8,'c')]

f = [(1,2), (3,4), (5,6)]
g = [math.sqrt(x*x+y*y)        # f = [2.23606, 5.0, 7.81024]
     for x,y in f]
```

Declarative Programming

List comprehensions and generator expressions are strongly tied to operations found in declarative languages. In fact, the origin of these features is loosely derived from ideas in mathematical set theory. For example, when you write a statement such as [x*x for x in a if x > 0], it's somewhat similar to specifying a set such as { x2 | x εa, x > 0 }.

Instead of writing programs that manually iterate over data, you can use these declarative features to structure programs as a series of computations that simply operate on all of the data all at once. For example, suppose you had a file "portfolio.txt" containing stock portfolio data like this:

```
AA 100 32.20
IBM 50 91.10
CAT 150 83.44
MSFT 200 51.23
GE 95 40.37
MSFT 50 65.10
IBM 100 70.44
```

Here is a declarative style program that calculates the total cost by summing up the second column multiplied by the third column:

```
lines  = open("portfolio.txt")
fields = (line.split() for line in lines)
print(sum(float(f[1]) * float(f[2]) for f in fields))
```

In this program, we really aren't concerned with the mechanics of looping line-by-line over the file. Instead, we just declare a sequence of calculations to perform on all of the data. Not only does this approach result in highly compact code, but it also tends to run faster than this more traditional version:

```
total = 0
for line in open("portfolio.txt"):
    fields = line.split()
    total += float(fields[1]) * float(fields[2])
print(total)
```

The lambda Operator

Anonymous functions in the form of an expression can be created using the lambda statement:

lambda args : expression

args is a comma-separated list of arguments, and expression is an expression involving those arguments. Here's an example:

```
a = lambda x,y : x+y
r = a(2,3)              # r gets 5
```

The code defined with lambda must be a valid expression. Multiple statements and other non-expression statements, such as for and while, cannot appear in a lambda statement. lambda expressions follow the same scoping rules as functions.

The primary use of lambda is in specifying short callback functions. For example, if you wanted to sort a list of names with case-insensitivity, you might write this:

names.sort(key=lambda n: n.lower())

Recursion

Recursive functions are easily defined. For example:

```
def factorial(n):
    if n <= 1: return 1
    else: return n * factorial(n - 1)
```

However, be aware that there is a limit on the depth of recursive function calls. The function **sys.getrecursionlimit()** returns the current maximum recursion depth, and the function **sys.setrecursionlimit()** can be used to change the value. The default value is 1000. Although it is possible to increase the value, programs are still limited by the stack size limits enforced by the host operating system. When the recursion depth is exceeded, a **RuntimeError** exception is raised. Python does not perform tail-recursion optimization that you often find in functional languages such as Scheme.

Recursion does not work as you might expect in generator functions and coroutines. For example, this code prints all items in a nested collection of lists:

```
def flatten(lists):
    for s in lists:
        if isinstance(s,list):
            flatten(s)
        else:
            print(s)

items = [[1,2,3],[4,5,[5,6]],[7,8,9]]
flatten(items)     # Prints 1 2 3 4 5 6 7 8 9
```

However, if you change the print operation to a yield, it no longer works. This is because the recursive call to flatten() merely creates a new generator object without actually iterating over it. Here's a recursive generator version that works:

```
def genflatten(lists):
    for s in lists:
        if isinstance(s,list):
            for item in genflatten(s):
                yield item
        else:
            yield item
```

Care should also be taken when mixing recursive functions and decorators. If a decorator is applied to a recursive function, all inner recursive calls now get routed through the decorated version. For example:

```
@locked
def factorial(n):
    if n <= 1: return 1
    else: return n * factorial(n - 1)  # Calls the wrapped version of factorial
```

If the purpose of the decorator was related to some kind of system management such as synchronization or locking, recursion is something probably best avoided.