3. The Python Language - Python in a Nutshell, 3rd Edition

Functions

Most statements in a typical Python program are part of functions. Code in a function body may be faster than at a module's top level, as covered in "Avoid exec and from … import *", so there are excellent practical reasons to put most of your code into functions, in addition to the advantages in clarity and readability. On the other hand, there are *no* disadvantages to organizing your code into functions versus leaving the code at module level.

A *function* is a group of statements that execute upon request. Python provides many built-in functions and allows programmers to define their own functions. A request to execute a function is known as a *function call*. When you call a function, you can pass arguments that specify data upon which the function performs its computation. In Python, a function always returns a result value, either `None` or a value that represents the results of the computation. Functions defined within `class` statements are also known as *methods*. We cover issues specific to methods in "Bound and Unbound Methods"; the general coverage of functions in this section, however, also applies to methods.

In Python, functions are objects (values), handled just like other objects. Thus, you can pass a function as an argument in a call to another function. Similarly, a function can return another function as the result of a call. A function, just like any other object, can be bound to a variable, can be an item in a container, and can be an attribute of an object. Functions can also be keys into a dictionary. For example, if you need to quickly find a function's inverse given the function, you could define a dictionary whose keys and values are functions and then make the dictionary bidirectional. Here's a small example of this idea, using some functions from module `math`, covered in "The math and cmath Modules", where we create a basic inverse mapping and add the inverse of each entry:

```
def make_inverse(inverse_dict):
    for f in list(inverse_dict):
        inverse_dict[inverse_dict[f]] = f
    return inverse_dict
inverse = make_inverse({sin:asin, cos:acos, tan:atan, log:exp})
```

The fact that functions are ordinary objects in Python is often expressed by saying that functions are *first-class* objects.

The def Statement

The `def` statement is the most common way to define a function. `def` is a single-clause compound statement with the following syntax:

```
def function_name(parameters):    statement(s)
```

`function_name` is an identifier. It is a variable that gets bound (or rebound) to the function object when `def` executes.

`parameters` is an optional list of identifiers that get bound to the values supplied when you call the function, which we refer to as *arguments* to distinguish between definitions (parameters) and calls (arguments). In the simplest case a function doesn't have any parameters, which means the function won't accept any arguments when you call it. In

this case, the function definition has empty parentheses after `function_name`, as must all calls to the function.

When a function does accept arguments, `parameters` contains one or more identifiers, separated by commas (`,`). In this case, each call to the function supplies values, known as *arguments*, corresponding to the parameters listed in the function definition. The parameters are local variables of the function (as we'll discuss later in this section), and each call to the function binds these local variables in the function namespace to the corresponding values that the caller, or the function definition, supplies as arguments.

The nonempty block of statements, known as the *function body*, does not execute when the `def` statement executes. Rather, the function body executes later, each time you call the function. The function body can contain zero or more occurrences of the `return` statement, as we'll discuss shortly.

Here's an example of a simple function that returns a value that is twice the value passed to it each time it's called:

```
def twice(x):
    return x*
2
```

Note that the argument can be anything that can be multiplied by two, so the function could be called with a number, string, list, or tuple as an argument, and it would return, in each case, a new value of the same type as the argument.

Parameters

Parameters (more pedantically, *formal parameters)* name, and sometimes specify a default value for, the values passed into a function call. Each of the names is bound inside a new local namespace every time the function is called; this namespace is destroyed when the function returns or otherwise exits (values returned by the function may be kept alive by the caller binding the function result). Parameters that are just identifiers indicate *positional parameters* (also known as *mandatory parameters*). Each call to the function must supply a corresponding value (argument) for each mandatory (positional) parameter.

In the comma-separated list of parameters, zero or more positional parameters may be followed by zero or more *named parameters* (also known as *optional parameters*, and sometimes—confusingly, since they do not involve any keyword—as *keyword parameters*), each with the syntax:

```
identifier=expression
```

The `def` statement evaluates each such `expression` and saves a reference to the expression's value, known as the *default value* for the parameter, among the attributes of the function object. When a function call does not supply an argument corresponding to a named parameter, the call binds the parameter's identifier to its default value for that execution of the function.

Note that Python computes each default value precisely once, when the `def` statement evaluates, *not* each time the resulting function gets called. In particular, this means that exactly the *same* object, the default value, gets bound to the named parameter whenever the caller does not supply a corresponding argument.

Beware using mutable default values

A mutable default value, such as a list, can be altered each time you call the function without an explicit argument corresponding to the respective parameter. This is usually not the behavior you want; see the following detailed

explanation.

Things can be tricky when the default value is a mutable object and the function body alters the parameter. For example:

```
def f(x, y=[]):    y.append(x)    return y, id(y)print(f(23))                # prints:
([23],                                                ([23, 42],
4302354376)       print(f(42))                # prints: 4302354376)
```

The second `print` prints `[23, 42]` because the first call to `f` altered the default value of `y`, originally an empty list `[]`, by appending `23` to it. The `id` values confirm that both calls return the same object. If you want `y` to be bound to a new empty list object each time you call `f` with a single argument, use the following idiom instead:

```
def f(x, y=None):    if y is None: y = []    y.append(x)    return y, id(y)print(f(23))
                          ([23],
               # prints: 4302354376)       prinf(f(42))               # prints:
([42],
4302180040)
```

Of course, there are cases in which you explicitly want to alter a parameter's default value, most often for caching purposes, as in the following example:

```
def cached_compute(x, _cache={}):
    if x not in _cache:
        _cache[x] = costly_computation(x
)
    return _cache[x]
```

Such caching behavior (also known as *memoization*), however, is usually best obtained by decorating the underlying function with `functools.lru_cache`, covered in Table 7-4.

At the end of the parameters, you may optionally use either or both of the special forms `*args` and `**kwds`. There is nothing special about the names—you can use any identifier you want in each special form; `args` and `kwds` or `kwargs`, as well as `a` and `k`, are just popular identifiers to use in these roles. If both forms are present, the form with two asterisks must come second.

`*args` specifies that any extra positional arguments to a call (i.e., those positional arguments not explicitly specified in the signature, defined in "Function signatures" below) get collected into a (possibly empty) tuple, and bound in the call namespace to the name `args`. Similarly `**kwds` specifies that any extra named arguments (i.e., those named arguments not explicitly specified in the signature, defined in "Function signatures" below) get collected into a (possibly empty) dictionary whose items are the names and values of those arguments, and bound to the name `kwds` in the function call namespace (we cover positional and named *arguments* in "Calling Functions").

For example, here's a function that accepts any number of positional arguments and returns their sum (also showing the use of an identifier other than *`*args`*):

```
def sum_args(*numbers):    return sum(numbers)print(sum_args(23, 42))
# prints: 65
```

Function signatures

The number of parameters of a function, together with the parameters' names, the number of mandatory parameters, and the information on whether (at the end of the parameters) either or both of the single- and double-asterisk special forms are present, collectively form a specification known as the function's *signature*. A function's signature defines the ways in which you can call the function.

"Keyword-only" Parameters (v3 Only)

In v3 only, a `def` statement also optionally lets you specify parameters that *must*, when you call the function, correspond to named arguments of the form *`identifier=expression`* (see "Kinds of arguments") if they are present at all. These are known as (confusingly, since keywords have nothing to do with them) as *keyword-only parameters*. If these parameters appear, they must come after `*args`(if any) and before `**kwargs` (if any).

Each keyword-only parameter can be specified in the parameter list as either a simple identifier (in which case it's mandatory, when calling the function, to pass the corresponding named argument), or in `identifier=default` form (in which case it's optional, when calling the function, whether to pass the corresponding argument—but, when you pass it, you *must* pass it as a named argument, not as a positional one). Either form can be used for each different keyword-only parameter. It's more usual and readable to have simple identifiers, if any, at the start of the keyword-only parameter specifications, and *`identifier=default`* forms, if any, following them, though this is not a requirement of the language.

All the keyword-only parameter specifications come *after* the special form `*args`, if the latter is present. If that special form is not present, then the start of the sequence of keyword-only parameter specifications is a null parameter, consisting solely of an `*` (asterisk), to which no argument corresponds. For example:

```
                        # b and c are keyword-
def f(a, *, b, c=56):   only
    return a, b, c
            # returns (12, 34, 56)—c's optional, having a
f(12,b=34)  default
f(12)       # raises a TypeError exception
# error message is: missing 1 required keyword-only argument:
'b'
```

If you also specify the special form *`**kwds`*, it must come at the end of the parameter list (after the keyword-only parameter specifications, if any). For example:

```
                           # b is keyword-
def g(x, *a, b=23, **k):   only
    return x, a, b, k
g(1, 2, 3, c=99)
# returns (1, (2, 3), 23, {'c':
99})
```

Attributes of Function Objects

The `def` statement sets some attributes of a function object. Attribute `__name__` refers to the identifier string given as the function name in the `def`. You may rebind the attribute to any string value, but trying to unbind it raises an exception. Attribute `__defaults__`, which you may freely rebind or unbind, refers to the tuple of default values for

optional parameters (an empty tuple, if the function has no optional parameters).

Docstrings

Another function attribute is the *documentation string*, also known as the *docstring*. You may use or rebind a function's docstring attribute as `__doc__`. If the first statement in the function body is a string literal, the compiler binds that string as the function's docstring attribute. A similar rule applies to classes (see "Class documentation strings") and modules (see "Module documentation strings"). Docstrings usually span multiple physical lines, so you normally specify them in triple-quoted string literal form. For example:

```
def sum_args(*numbers):

"""Return the sum of multiple numerical
arguments.

    The arguments are zero or more numbers.
    The result is their
sum.

"""
    return sum(numbers)
```

Documentation strings should be part of any Python code you write. They play a role similar to that of comments, but their applicability is wider, since they remain available at runtime (unless you run your program with `python -OO`, as covered in "Command-Line Syntax and Options"). Development environments and tools can use docstrings from function, class, and module objects to remind the programmer how to use those objects. The `doctest` module (covered in "The doctest Module") makes it easy to check that sample code present in docstrings is accurate and correct, and remains so, as the code and docs get edited and maintained.

To make your docstrings as useful as possible, respect a few simple conventions. The first line of a docstring should be a concise summary of the function's purpose, starting with an uppercase letter and ending with a period. It should not mention the function's name, unless the name happens to be a natural-language word that comes naturally as part of a good, concise summary of the function's operation. If the docstring is multiline, the second line should be empty, and the following lines should form one or more paragraphs, separated by empty lines, describing the function's parameters, preconditions, return value, and side effects (if any). Further explanations, bibliographical references, and usage examples (which you should check with `doctest`) can optionally follow toward the end of the docstring.

Other attributes of function objects

In addition to its predefined attributes, a function object may have other arbitrary attributes. To create an attribute of a function object, bind a value to the appropriate attribute reference in an assignment statement after the `def` statement executes. For example, a function could count how many times it gets called:

```
def counter():
    counter.count += 1
    return counter.
count
counter.count = 0
```

Note that this is *not* common usage. More often, when you want to group together some state (data) and some behavior (code), you should use the object-oriented mechanisms covered in Chapter 4. However, the ability to associate arbitrary attributes with a function can sometimes come in handy.

Function Annotations and Type Hints (v3 Only)

In v3 only, every parameter in a `def` clause can be *annotated* with an arbitrary expression—that is, wherever within the `def`'s parameter list you can use an identifier, you can alternatively use the form *`identifier:expression`*, and the expression's value becomes the *annotation* for that parameter name.

You can annotate the return value of the function, using the form *`->expression`* between the `)` of the `def` clause and the `:` that ends the `def` clause; the expression's value becomes the annotation for name `'return'`. For example:

```
>>> def f(a:'foo', b)->'bar': pass... >>> f.__annotations__{'a': 'foo', 'return': 'bar'
}
```

As shown in this example, the `__annotations__` attribute of the function object is a `dict` mapping each annotated identifier to the respective annotation.

You can use annotations for whatever purpose you wish: Python itself does nothing with them, except constructing the `__annotations__` attribute.

The intention is to let future, hypothetical third-party tools leverage annotations to perform more thorough static checking on annotated functions. (If you want to experiment with such static checks, we recommend you try the mypy project.)

Type hints (Python 3.5 only)

To further support hypothetical third-party tools, Python 3.5 has introduced a complicated set of commenting conventions, and a new *provisional* module called `typing` in the standard library, to standardize how annotations are to be used to denote typing hints (again, Python does nothing with the information, but third-party tools may eventually be developed to leverage this now-standardized information).

A provisional module is one that might change drastically, or even disappear, as early as the next feature release (so, in the future, Python 3.7 might not have the module `typing` in its standard library, or the module's contents might be different). It is not yet intended for use "in production," but rather just for experimental purposes.

As a consequence, we do not further cover type hints in this book; see PEP 484 and the many links from it for much, much more. (The mypy third-party project does fully support PEP 484–compliant v3 Python code).

New in 3.6: Type annotations

Although the conventions defined in PEP 484 could help third-party static analysis tools operate on Python code, the PEP does not include syntax support for explicitly marking variables as being of a specific type; such type annotations were embedded in the form of comments. PEP 526 now defines syntax for variable type annotations.

Guido van Rossum has cited anecdotal evidence that type annotations can be helpful, for example, when engineering large legacy codebases. They may be unnecessarily complicated for beginners, but they remain completely optional and so can be taught as required to those with some Python experience. While you can still use PEP 484–style comments, it seems likely that PEP 526 annotations will supersede them. (The mypy third-party

project, at the time of this writing, only partially supports PEP 526–compliant Python 3.6 code.)

The return Statement

The `return` statement in Python can exist only inside a function body, and can optionally be followed by an expression. When `return` executes, the function terminates, and the value of the expression is the function's result. A function returns `None` if it terminates by reaching the end of its body or by executing a `return` statement that has no expression (or, of course, by explicitly executing `return None`).

Good style in return statements

As a matter of good style, *never* write a `return` statement without an expression at the end of a function body. When some `return` statements in a function have an expression, then all `return` statements in the function should have an expression. `return None` should only ever be written explicitly to meet this style requirement. Python does not enforce these stylistic conventions, but your code is clearer and more readable when you follow them.

Calling Functions

A function call is an expression with the following syntax:

```
function_object(arguments)
```

`function_object` may be any reference to a function (or other callable) object; most often, it's the function's name. The parentheses denote the function-call operation itself. `arguments`, in the simplest case, is a series of zero or more expressions separated by commas (`,`), giving values for the function's corresponding parameters. When the function call executes, the parameters are bound to the argument values, the function body executes, and the value of the function-call expression is whatever the function returns.

Don't forget the trailing () to call a function

Just *mentioning* a function (or other callable object) does *not*, per se, call it. To *call* a function (or other object) without arguments, you *must* use `()` after the function's name (or other reference to the callable object).

The semantics of argument passing

In traditional terms, all argument passing in Python is *by value* (although, in modern terminology, to say that argument passing is *by object reference* is more precise and accurate; some, but not all, of this book's authors also like the synonym *call by sharing*). For example, if you pass a variable as an argument, Python passes to the function the object (value) to which the variable currently refers, not the variable itself, and this value is bound to the parameter name in the function call namespace. Thus, a function cannot rebind the caller's variables. However, if you pass a mutable object as an argument, the function may request changes to that object, because Python passes the object itself, not a copy. Rebinding a variable and mutating an object are totally disjoint concepts. For example:

```
def f(x, y):    x = 23    y.append(42)a = 77b = [99]f(a, b)print(a, b)
          77 [99,
# prints: 42]
```

`print` shows that `a` is still bound to `77`. Function `f`'s rebinding of its parameter `x` to `23` has no effect on `f`'s caller, nor, in particular, on the binding of the caller's variable that happened to be used to pass `77` as the parameter's value. However, `print` also shows that `b` is now bound to `[99, 42]`. `b` is still bound to the same list object as before the call, but that object has mutated, as `f` has appended `42` to that list object. In either case, `f` has not altered the caller's bindings, nor can `f` alter the number `77`, since numbers are immutable. However, `f` can alter a list object, since list objects are mutable. In this example, `f` mutates the list object that the caller passes to `f` as the second argument by calling the object's `append` method.

Kinds of arguments

Arguments that are just expressions are known as *positional arguments*. Each positional argument supplies the value for the parameter that corresponds to it by position (order) in the function definition. There is no requirement for positional arguments to match positional parameters—if there are more positional arguments that positional parameters, the additional arguments are bound by position to named parameters, if any, and then (if the function's signature includes a `*args` special form) to the tuple bound to `args`. For example:

```
def f(a, b, c=23, d=42, *x):    print(a, b, c, d, x)f(1,2,3,4,5,6)  # prints
(1, 2, 3, 4, (5,
6))
```

In a function call, zero or more positional arguments may be followed by zero or more *named arguments* (sometimes confusingly known as *keyword arguments*), each with the following syntax:

```
identifier=expression
```

In the absence of any `**kwds` parameter, the `identifier` must be one of the parameter names used in the `def` statement for the function. The `expression` supplies the value for the parameter of that name. Many built-in functions do not accept named arguments: you must call such functions with positional arguments only. However, functions coded in Python accept named as well as positional arguments, so you may call them in different ways. Positional parameters can be matched by named arguments, in the absence of matching positional arguments.

A function call must supply, via a positional or a named argument, exactly one value for each mandatory parameter, and zero or one value for each optional parameter. For example:

```
def divide(divisor, dividend):    return dividend // divisorprint(divide(12, 94))
                      # prints: 7print(divide(dividend=94, divisor=12))
# prints: 7
```

As you can see, the two calls to `divide` are equivalent. You can pass named arguments for readability purposes whenever you think that identifying the role of each argument and controlling the order of arguments enhances your code's clarity.

A common use of named arguments is to bind some optional parameters to specific values, while letting other optional parameters take default values:

```
def f(middle, begin='init', end='finis'):    return begin+middle+endprint(f('tini', end
='''))                    # prints: inittini
```

Using named argument `end=''`, the caller specifies a value (the empty string `''`) for `f`'s third parameter, `end`, and still lets `f`'s second parameter, `begin`, use its default value, the string `'init'`. Note that you may pass the arguments as positional arguments, even when the parameters are named; for example, using the preceding function:

```
print(f('a','c','t'))                          # prints: cat
```

At the end of the arguments in a function call, you may optionally use either or both of the special forms `*seq` and `**dct`. If both forms are present, the form with two asterisks must be last. `*seq` passes the items of `seq` to the function as positional arguments (after the normal positional arguments, if any, that the call gives with the usual syntax). `seq` may be any iterable. `**dct` passes the items of `dct` to the function as named arguments, where `dct` must be a dictionary whose keys are all strings. Each item's key is a parameter name, and the item's value is the argument's value.

Sometimes you want to pass an argument of the form `*seq` or `**dct` when the parameters use similar forms, as covered earlier in "Parameters". For example, using the function `sum_args` defined in that section (and shown again here), you may want to print the sum of all the values in dictionary `d`. This is easy with `*seq`:

```
def sum_args(*numbers):
    return sum(numbers)
print(sum_args(*d.values()))
```

(Of course, `print(sum(d.values()))` would be simpler and more direct.)

You may also pass arguments `*seq` or `**dct` when calling a function that does not use the corresponding forms in its signature. In that case, of course, you must ensure that iterable `seq` has the right number of items, or, respectively, that dictionary `dct` uses the right identifier strings as its keys; otherwise, the call operation raises an exception. As noted in ""Keyword-only" Parameters (v3 Only)", keyword-only parameters in v3 *cannot* be matched with a positional argument (unlike named parameters); they must match a named argument, explicit or passed via `**dct`.

In v3, a function call may have zero or more occurrences of `*seq` and/or `**dct`, as specified in PEP 448.

Namespaces

A function's parameters, plus any names that are bound (by assignment or by other binding statements, such as `def`) in the function body, make up the function's *local namespace*, also known as *local scope*. Each of these variables is known as a *local variable* of the function.

Variables that are not local are known as *global variables* (in the absence of nested function definitions, which we'll discuss shortly). Global variables are attributes of the module object, as covered in "Attributes of module objects". Whenever a function's local variable has the same name as a global variable, that name, within the function body, refers to the local variable, not the global one. We express this by saying that the local variable *hides* the global variable of the same name throughout the function body.

The global statement

By default, any variable that is bound within a function body is a local variable of the function. If a function needs to bind or rebind some global variables (*not* a good practice!), the first statement of the function's body must be:

```
global identifiers
```

where `identifiers` is one or more identifiers separated by commas (`,`). The identifiers listed in a `global` statement refer to the global variables (i.e., attributes of the module object) that the function needs to bind or rebind. For example, the function `counter` that we saw in "Other attributes of function objects" could be implemented using `global` and a global variable, rather than an attribute of the function object:

```
_count = 0
def counter():
    global
_count
    _count += 1
    return
_count
```

Without the `global` statement, the `counter` function would raise an `UnboundLocalError` exception because `_count` would then be an uninitialized (unbound) local variable. While the `global` statement enables this kind of programming, this style is inelegant and unadvisable. As we mentioned earlier, when you want to group together some state and some behavior, the object-oriented mechanisms covered in Chapter 4 are usually best.

Eschew global

Never use `global` if the function body just *uses* a global variable (including mutating the object bound to that variable, when the object is mutable). Use a `global` statement only if the function body *rebinds* a global variable (generally by assigning to the variable's name). As a matter of style, don't use `global` unless it's strictly necessary, as its presence causes readers of your program to assume the statement is there for some useful purpose. In particular, never use `global` except as the first statement in a function body.

Nested functions and nested scopes

A `def` statement within a function body defines a *nested function*, and the function whose body includes the `def` is known as an *outer function* to the nested one. Code in a nested function's body may access (but not rebind) local variables of an outer function, also known as *free variables* of the nested function.

The simplest way to let a nested function access a value is often not to rely on nested scopes, but rather to pass that value explicitly as one of the function's arguments. If necessary, the argument's value can be bound at nested function `def` time, by using the value as the default for an optional argument. For example:

```
def percent1(a, b, c):
    def pc(x, total=a+b+c):
        return (x*100.0) / total
    print('Percentages are:', pc(a), pc(b), pc(c
))
```

Here's the same functionality using nested scopes:

```
def percent2(a, b, c):
    def pc(x):
        return (x*100.0) / (a+b+c)
    print('Percentages are:', pc(a), pc(b), pc(c
))
```

In this specific case, `percent1` has at least a tiny advantage: the computation of `a+b+c` happens only once, while `percent2`'s inner function `pc` repeats the computation three times. However, when the outer function rebinds local variables between calls to the nested function, repeating the computation can be necessary: be aware of both approaches, and choose the appropriate one case by case.

A nested function that accesses values from outer local variables is also known as a *closure*. The following example shows how to build a closure:

```
def make_adder(augend):
    def add(addend):
        return addend+
augend
    return add
```

Closures are sometimes an exception to the general rule that the object-oriented mechanisms covered in Chapter 4 are the best way to bundle together data and code. When you need specifically to construct callable objects, with some parameters fixed at object-construction time, closures can be simpler and more effective than classes. For example, the result of `make_adder(7)` is a function that accepts a single argument and adds `7` to that argument. An outer function that returns a closure is a "factory" for members of a family of functions distinguished by some parameters, such as the value of argument `augend` in the previous example, and may often help you avoid code duplication.

In v3 only, the `nonlocal` keyword acts similarly to `global`, but it refers to a name in the namespace of some lexically surrounding function. When it occurs in a function definition nested several levels deep, the compiler searches the namespace of the most deeply nested containing function, then the function containing that one, and so on, until the name is found or there are no further containing functions, in which case the compiler raises an error.

Here's a v3-only nested-functions approach to the "counter" functionality we implemented in previous sections using a function attribute, then a global variable:

```
def make_counter():    count = 0    def counter():        nonlocal count        count +
= 1        return count    return counterc1 = make_counter()c2 = make_counter()print(
                                1 2                                     1
c1(), c1(), c1())    # prints: 3    print(c2(), c2())          # prints: 2  print(c1(),
                              4 3
 c2(), c1())    # prints: 5
```

A key advantage of this approach versus the previous ones is that nested functions, just like an OOP approach, let you make independent counters, here `c1` and `c2`—each closure keeps its own state and doesn't interfere with the other.

This example is v3-only because it requires `nonlocal`, since the inner function `counter` needs to rebind free variable `count`. In v2, you would need a somewhat nonintuitive trick to achieve the same effect without rebinding a free variable:

```
def make_counter():
    count = [0]
    def counter():
        count[0] += 1
        return count[0
]
    return counter
```

Clearly, the v3-only version is more readable, since it doesn't require any trick.

lambda Expressions

If a function body is a single `return expression` statement, you may choose to replace the function with the special `lambda` expression form:

```
lambda parameters: expression
```

A `lambda` expression is the anonymous equivalent of a normal function whose body is a single `return` statement. Note that the `lambda` syntax does not use the `return` keyword. You can use a `lambda` expression wherever you could use a reference to a function. `lambda` can sometimes be handy when you want to use an extremely simple function as an argument or return value. Here's an example that uses a `lambda` expression as an argument to the built-in `filter` function (covered in Table 7-2):

```
a_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
low = 3
high = 7
filter(lambda x: low<=x<high, a_list)
# returns: [3, 4, 5,
6]
```

(In v3, `filter` returns an iterator, not a list.) Alternatively, you can always use a local `def` statement to give the function object a name, then use this name as an argument or return value. Here's the same `filter` example using a local `def` statement:

```
a_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
def within_bounds(value, low=3, high=7):
    return low<=value><high
filter(within_bounds, a_list)
# returns: [3, 4, 5,
6]
```

While `lambda` can at times be handy, `def` is usually better: it's more general, and makes the code more readable, as you can choose a clear name for the function.

Generators

When the body of a function contains one or more occurrences of the keyword `yield`, the function is known as a *generator*, or more precisely a *generator function*. When you call a generator, the function body does not execute.

Instead, the generator function returns a special iterator object, known as a *generator object* (sometimes, somewhat confusingly, also referred to as just "a generator"), that wraps the function body, its local variables (including its parameters), and the current point of execution, which is initially the start of the function.

When you call `next` on a generator object, the function body executes from the current point up to the next `yield`, which takes the form:

```
yield expression
```

A bare `yield` without the expression is also legal, and equivalent to `yield None`. When a `yield` executes, the function execution is "frozen," with current point of execution and local variables preserved, and the expression following `yield` returned as the result of `next`. When you call `next` again, execution of the function body resumes where it left off, again up to the next `yield`. When the function body ends or executes a `return` statement, the iterator raises a `StopIteration` exception to indicate that the iteration is finished. In v2, `return` statements in a generator cannot contain expressions; in v3, the expression after `return`, if any, is an argument to the resulting `StopIteration`.

`yield` is an expression, not a statement. When you call `g.send(value)` on a generator object `g`, the value of the `yield` is `value`; when you call `next(g)`, the value of the `yield` is `None`. We cover this in "Generators as near-coroutines": it's the elementary building block to implement coroutines in Python.

A generator function is often a very handy way to build an iterator. Since the most common way to use an iterator is to loop on it with a `for` statement, you typically call a generator like this (with the call to `next` implicit in the `for` statement):

```
for avariable in somegenerator(arguments):
```

For example, say that you want a sequence of numbers counting up from `1` to `N` and then down to `1` again. A generator can help:

```
def updown(N):    for x in range(1, N):        yield x    for x in range(N, 0, -1):
                                                                1 2 3 2
        yield xfor i in updown(3): print(i)                  # prints: 1
```

Here is a generator that works somewhat like built-in `range`, but returns an iterator on floating-point values rather than on integers:

```
def frange(start, stop, stride=1.0):
    while start < stop:
        yield start
        start += stride
```

This `frange` example is only *somewhat* like `range` because, for simplicity, it makes arguments `start` and `stop` mandatory, and assumes `step` is positive.

Generators are more flexible than functions that return lists. A generator may return an *unbounded* iterator, meaning one that yields an infinite stream of results (to use only in loops that terminate by other means, e.g., via a `break` statement). Further, a generator-object iterator performs *lazy evaluation*: the iterator computes each successive item

only when and if needed, just in time, while the equivalent function does all computations in advance and may require large amounts of memory to hold the results list. Therefore, if all you need is the ability to iterate on a computed sequence, it is most often best to compute the sequence in a generator, rather than in a function that returns a list. If the caller needs a list of all the items produced by some bounded generator `g(arguments)`, the caller can simply use the following code to explicitly request that a list be built:

```
resulting_list = list(g(arguments))
```

yield from (v3-only)

In v3 only, to improve execution efficiency and clarity when multiple levels of iteration are yielding values, you can use the form `yield from expression`, where `expression` is iterable. This yields the values from `expression` one at a time into the calling environment, avoiding the need to `yield` repeatedly. The `updown` generator defined earlier can be simplified in v3:

```
def updown(N):
    yield from range(1, N)
    yield from range(N, 0, -1)
for i in updown(3): print(i)
# prints: 1 2 3 2
1
```

Moreover, the ability to use `yield from` means that, in v3 only, generators can be used as full-fledged *coroutines*, as covered in Chapter 18.

Generator expressions

Python offers an even simpler way to code particularly simple generators: *generator expressions*, commonly known as *genexps*. The syntax of a genexp is just like that of a list comprehension (as covered in "List comprehensions"), except that a genexp is within parentheses (`()`) instead of brackets (`[]`). The semantics of a genexp are the same as those of the corresponding list comprehension, except that a genexp produces an iterator yielding one item at a time, while a list comprehension produces a list of all results in memory (therefore, using a genexp, when appropriate, saves memory). For example, to sum the squares of all single-digit integers, you could code `sum([x*x for x in range(10)])`; however, you can express this even better, coding it as `sum(x*x for x in range(10))` (just the same, but omitting the brackets), and obtain exactly the same result while consuming less memory. Note that the parentheses that indicate the function call also "do double duty" and enclose the genexp (no need for extra parentheses).

Generators as near-coroutines

In all modern Python versions (both v2 and v3), generators are further enhanced, with the possibility of receiving a value (or an exception) back from the caller as each `yield` executes. This lets generators implement coroutines, as explained in PEP 342. The main change from older versions of Python is that `yield` is not a statement, but an expression, so it has a value. When a generator resumes via a call to `next` on it, the corresponding `yield`'s value is `None`. To pass a value `x` into some generator `g` (so that `g` receives `x` as the value of the `yield` on which it's suspended), instead of calling `next(g)`, call `g.send(x)` (`g.send(None)` is just like `next(g)`).

Other modern enhancements to generators have to do with exceptions, and we cover them in "Generators and

Exceptions".

Recursion

Python supports recursion (i.e., a Python function can call itself), but there is a limit to how deep the recursion can be. By default, Python interrupts recursion and raises a `RecursionLimitExceeded` exception (covered in "Standard Exception Classes") when it detects that the stack of recursive calls has gone over a depth of 1,000. You can change the recursion limit with function `setrecursionlimit` of module `sys`, covered in Table 7-3.

However, changing the recursion limit does not give you unlimited recursion; the absolute maximum limit depends on the platform on which your program is running, particularly on the underlying operating system and C runtime library, but it's typically a few thousand levels. If recursive calls get too deep, your program crashes. Such runaway recursion, after a call to `setrecursionlimit` that exceeds the platform's capabilities, is one of the very few ways a Python program can crash—really crash, hard, without the usual safety net of Python's exception mechanisms. Therefore, beware "fixing" a program that is getting `RecursionLimitExceeded` exceptions by raising the recursion limit with `setrecursionlimit`. Most often, you're better advised to look for ways to remove the recursion or, at least, limit the depth of recursion that your program needs.

Readers who are familiar with Lisp, Scheme, or functional programming languages must in particular be aware that Python does *not* implement the optimization of "tail-call elimination," which is so important in these languages. In Python, any call, recursive or not, has the same cost in terms of both time and memory space, dependent only on the number of arguments: the cost does not change, whether the call is a "tail-call" (meaning that the call is the last operation that the caller executes) or any other, nontail call. This makes recursion removal even more important.

For example, consider a classic use for recursion: "walking a tree." Suppose you have a "tree" data structure where each node is a tuple with three items: first, a payload value as item 0; then, as item 1, a similar tuple for the left-side descendant, or `None`; and similarly as item 2, for the right-side descendant. A simple example might be:

```
(23, (42, (5, None, None), (55, None, None)), (94,
None,None))
```

to represent the tree where the payloads are:

```
          23
       42
94
      5  55
```

We want to write a generator function that, given the root of such a tree, "walks the tree," yielding each payload in top-down order. The simplest approach is clearly recursion; in v3:

```
def rec(t):
    yield t[0]
    for i in (1, 2):
        if t[i] is not None: yield from rec(t[i
])
```

However, if a tree is very deep, recursion becomes a problem. To remove recursion, we just need to handle our own stack—a list used in last-in, first-out fashion, thanks to its `append` and `pop` methods. To wit, in either v2 or v3:

```
def norec(t):
    stack = [t]
    while stack:
        t = stack.pop()
        yield t[0]
        for i in (2, 1):
            if t[i] is not None: stack.append(t[i
])
```

The only small issue to be careful about, to keep exactly the same order of `yield`s as `rec`, is switching the `(1, 2)` index order in which to examine descendants, to `(2, 1)`, adjusting to the reverse (last-in, first-out) behavior of `stack`.