You can also have a target with multiple identifiers, as with an unpacking assignment. In this case, the iterator's items must then be iterables, each with exactly as many items as there are identifiers in the target. For example, when *d* is a dictionary, this is a typical way to loop on the items (key/value pairs) in *d*:

```
for key, value in d.items( ):
    if not key or not value:       # keep only true keys and values
        del d[key]
```

The `items` method returns a list of key/value pairs, so we can use a `for` loop with two identifiers in the target to unpack each item into `key` and `value`.

When an iterator has a mutable underlying object, you must not alter that object during a `for` loop on it. For example, the previous example cannot use `iteritems` instead of `items`. `iteritems` returns an iterator whose underlying object is `d`, so the loop body cannot mutate `d` (by executing `del d[key]`). `items` returns a list so that `d` is not the underlying object of the iterator; therefore, the loop body can mutate `d`. Specifically:

- When looping on a list, do not insert, append, or delete items (rebinding an item at an existing index is OK).
- When looping on a dictionary, do not add or delete items (rebinding the value for an existing key is OK).
- When looping on a set, do not add or delete items (no alteration is permitted).

The control variable may be rebound in the loop body but is rebound again to the next item in the iterator at the next iteration of the loop. The loop body does not execute at all if the iterator yields no items. In this case, the control variable is not bound or rebound in any way by the `for` statement. If the iterator yields at least one item, however, when the loop statement terminates, the control variable remains bound to the last value to which the loop statement has bound it. The following code is therefore correct, as long as `someseq` is not empty:

```
for x in someseq:
    process(x)
print "Last item processed was", x
```

#### Iterators

An *iterator* is an object *i* such that you can call *i*`.next( )` with no arguments. *i*`.next( )` returns the next item of iterator *i* or, when iterator *i* has no more items, raises a `StopIteration` exception. When you write a class (see "Classes and Instances" on page 82), you can allow instances of the class to be iterators by defining such a method `next`. Most iterators are built by implicit or explicit calls to built-in function `iter`, covered in `iter` on page 163. Calling a generator also returns an iterator, as we'll discuss in "Generators" on page 78.

The `for` statement implicitly calls `iter` to get an iterator. The following statement:

```
for x in c:
    statement(s)
```

is exactly equivalent to:

```
_temporary_iterator = iter(c)
while True:
    try: x = _temporary_iterator.next( )
    except StopIteration: break
    statement(s)
```

where *_temporary_iterator* is some arbitrary name that is not used elsewhere in the current scope.

Thus, if `iter(c)` returns an iterator *i* such that *i*`.next( )` never raises `StopIteration` (an *unbounded iterator*), the loop `for` *x* `in` *c* never terminates (unless the statements in the loop body include suitable `break` or `return` statements, or raise or propagate exceptions). `iter(c)`, in turn, calls special method *c*`.__iter__( )` to obtain and return an iterator on *c*. I'll talk more about the special method `__iter__` in `__iter__` on page 112.

Many of the best ways to build and manipulate iterators are found in standard library module `itertools`, covered in "The itertools Module" on page 183.

#### range and xrange

Looping over a sequence of integers is a common task, so Python provides built-in functions `range` and `xrange` to generate and return integer sequences. The simplest way to loop *n* times in Python is:

```
for i in xrange(n):
    statement(s)
```

`range(`*x*`)` returns a list whose items are consecutive integers from `0` (included) up to *x* (excluded). `range(`*x*`, `*y*`)` returns a list whose items are consecutive integers from *x* (included) up to *y* (excluded). The result is the empty list if *x* is greater than or equal to *y*. `range(`*x*`, `*y*`, `*step*`)` returns a list of integers from *x* (included) up to *y* (excluded), such that the difference between each two adjacent items in the list is *step*. If *step* is less than `0`, `range` counts down from *x* to *y*. `range` returns the empty list when *x* is greater than or equal to *y* and *step* is greater than `0`, or when *x* is less than or equal to *y* and *step* is less than `0`. When *step* equals `0`, `range` raises an exception.

While `range` returns a normal list object, usable for all purposes, `xrange` returns a special-purpose object, specifically intended for use in iterations like the `for` statement shown previously (unfortunately, to keep backward compatibility with old versions of Python, `xrange` does not return an iterator, as would be natural in today's Python; however, you can easily obtain such an iterator, if you need one, by calling `iter(xrange(...))`). The special-purpose object `xrange` returns consumes less memory (for wide ranges, *much* less memory) than the list object `range` returns, but the overhead of looping on the special-purpose object is slightly higher than that of looping on a list. Apart from performance and memory consumption issues, you can use `range` wherever you could use `xrange`, but not vice versa. For example:

```
>>> print range(1, 5)
[1, 2, 3, 4]
>>> print xrange(1, 5)
xrange(1, 5)
```

Here, range returns a perfectly ordinary list, which displays quite normally, but xrange returns a special-purpose object, which displays in its own special way.

### List comprehensions

A common use of a for loop is to inspect each item in an iterable and build a new list by appending the results of an expression computed on some or all of the items. The expression form known as a *list comprehension* lets you code this common idiom concisely and directly. Since a list comprehension is an expression (rather than a block of statements), you can use it wherever you need an expression (e.g., as an argument in a function call, in a return statement, or as a subexpression for some other expression).

A list comprehension has the following syntax:

```
[ expression for target in iterable lc-clauses ]
```

*target* and *iterable* are the same as in a regular for statement. You must enclose the *expression* in parentheses if it indicates a tuple.

*lc-clauses* is a series of zero or more clauses, each with one of the following forms:

```
for target in iterable
if expression
```

*target* and *iterable* in each for clause of a list comprehension have the same syntax and meaning as those in a regular for statement, and the *expression* in each if clause of a list comprehension has the same syntax and meaning as the *expression* in a regular if statement.

A list comprehension is equivalent to a for loop that builds the same list by repeated calls to the resulting list's append method. For example (assigning the list comprehension result to a variable for clarity):

```
result1 = [x+1 for x in some_sequence]
```

is the same as the for loop:

```
result2 = []
for x in some_sequence:
    result2.append(x+1)
```

Here's a list comprehension that uses an if clause:

```
result3 = [x+1 for x in some_sequence if x>23]
```

This list comprehension is the same as a for loop that contains an if statement:

```
result4 = []
for x in some_sequence:
    if x>23:
        result4.append(x+1)
```

And here's a list comprehension that uses a for clause:

```
result5 = [x+y for x in alist for y in another]
```

This is the same as a for loop with another for loop nested inside:

```
result6 = []
for x in alist:
    for y in another:
        result6.append(x+y)
```

As these examples show, the order of for and if in a list comprehension is the same as in the equivalent loop, but in the list comprehension, the nesting remains implicit.

## The break Statement

The break statement is allowed only inside a loop body. When break executes, the loop terminates. If a loop is nested inside other loops, a break in it terminates only the innermost nested loop. In practical use, a break statement is usually inside some clause of an if statement in the loop body so that break executes conditionally.

One common use of break is in the implementation of a loop that decides whether it should keep looping only in the middle of each loop iteration:

```
while True:                     # this loop can never terminate naturally
    x = get_next( )
    y = preprocess(x)
    if not keep_looping(x, y): break
    process(x, y)
```

## The continue Statement

The continue statement is allowed only inside a loop body. When continue executes, the current iteration of the loop body terminates, and execution continues with the next iteration of the loop. In practical use, a continue statement is usually inside some clause of an if statement in the loop body so that continue executes conditionally.

Sometimes, a continue statement can take the place of nested if statements within a loop. For example:

```
for x in some_container:
    if not seems_ok(x): continue
    lowbound, highbound = bounds_to_test( )
    if x<lowbound or x>=highbound: continue
    if final_check(x):
        do_processing(x)
```

This equivalent code does conditional processing without continue:

```
for x in some_container:
    if seems_ok(x):
        lowbound, highbound = bounds_to_test( )
        if lowbound <= x < highbound:
            if final_check(x):
                do_processing(x)
```

Both versions function identically, so which one you use is a matter of personal preference and style.

## The else Clause on Loop Statements

while and for statements may optionally have a trailing else clause. The statement or block under that else executes when the loop terminates *naturally* (at the end of the for iterator, or when the while loop condition becomes false), but not when the loop terminates *prematurely* (via break, return, or an exception). When a loop contains one or more break statements, you often need to check whether the loop terminates naturally or prematurely. You can use an else clause on the loop for this purpose:

```
for x in some_container:
    if is_ok(x): break            # item x is satisfactory, terminate loop
else:
    print "Warning: no satisfactory item was found in container"
    x = None
```

## The pass Statement

The body of a Python compound statement cannot be empty; it must always contain at least one statement. You can use a pass statement, which performs no action, as a placeholder when a statement is syntactically required but you have nothing to do. Here's an example of using pass in a conditional statement as a part of somewhat convoluted logic to test mutually exclusive conditions:

```
if condition1(x):
    process1(x)
elif x>23 or condition2(x) and x<5:
    pass                              # nothing to be done in this case
elif condition3(x):
    process3(x)
else:
    process_default(x)
```

Note that, as the body of an otherwise empty def or class statement, you may use a docstring, covered in "Docstrings" on page 72; if you do write a docstring, then you do not need to also add a pass statement, although you are still allowed to do so if you wish.

## The try and raise Statements

Python supports exception handling with the try statement, which includes try, except, finally, and else clauses. A program can explicitly raise an exception with the raise statement. As I discuss in detail in "Exception Propagation" on page 126, when an exception is raised, normal control flow of the program stops, and Python looks for a suitable exception handler.

## The with Statement

In Python 2.5, a with statement has been added as a more readable alternative to the try/finally statement. I discuss it in detail in "The with statement" on page 125.

# Functions

Most statements in a typical Python program are grouped and organized into functions (code in a function body may be faster than at a module's top level, as covered in "Avoiding exec and from…import *" on page 486, so there are excellent practical reasons to put most of your code into functions). 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*. Issues specific to methods are covered in "Bound and Unbound Methods" on page 91; the general coverage of functions in this section, however, also applies to methods.

In Python, functions are objects (values) that are handled 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, an item in a container, or 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" on page 365:

```
inverse = {sin:asin, cos:acos, tan:atan, log:exp}
for f in inverse.keys( ): inverse[inverse[f]] = f
```

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, known as *formal parameters* or just parameters, that get bound to the values supplied as arguments when the function is called. In the simplest case, a function doesn't have any formal parameters, which means the function doesn't take any arguments when it is called. In this case, the function definition has empty parentheses after *`function-name`*.

When a function does take 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 to the corresponding values that the caller supplies as arguments.

The nonempty sequence of statements, known as the *function body*, does not execute when the `def` statement executes. Rather, the function body executes later, each time the function is called. 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 double(x):
    return x*2
```

## Parameters

Formal parameters that are just identifiers indicate *mandatory parameters*. Each call to the function must supply a corresponding value (argument) for each mandatory parameter.

In the comma-separated list of parameters, zero or more mandatory parameters may be followed by zero or more *optional parameters*, where each optional parameter has 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 an optional parameter, the call binds the parameter's identifier to its default value for that execution of the function. Note that each default value gets computed when the `def` statement evaluates, *not* when the resulting function gets called. In particular, this means that the *same* object, the default value, gets bound to the optional parameter whenever the caller does not supply a corresponding argument. This 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
print f(23)                # prints: [23]
prinf f(42)                # prints: [23, 42]
```

The second `print` statement prints `[23, 42]` because the first call to `f` altered the default value of `y`, originally an empty list `[]`, by appending `23` to it. If you want `y` to be bound to a new empty list object each time `f` is called with a single argument, use the following style instead:

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

At the end of the parameters, you may optionally use either or both of the special forms *`identifier1` and **`identifier2`. If both forms are present, the form with two asterisks must be last. *`identifier1` specifies that any call to the function may supply any number of extra positional arguments, while **`identifier2` specifies that any call to the function may supply any number of extra named arguments (positional and named arguments are covered in "Calling Functions" on page 73). Every call to the function binds `identifier1` to a tuple whose items are the extra positional arguments (or the empty tuple, if there are none). Similarly, `identifier2` gets bound to a dictionary whose items are the names and values of the extra named arguments (or the empty dictionary, if there are none). Here's a function that accepts any number of positional arguments and returns their sum:

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

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.

## Attributes of Function Objects

The `def` statement sets some attributes of a function object. The attribute `func_name`, also accessible as `__name__`, refers to the identifier string given as the function name in the `def` statement. In Python 2.3, this is a read-only attribute (trying to rebind or unbind it raises a runtime exception); in Python 2.4, you may rebind the attribute to any string value, but trying to unbind it raises an exception. The attribute `func_defaults`, which you may freely rebind or unbind, refers to the tuple of default values for the optional parameters (or the 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 either `func_doc` or `__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" on page 85) and modules (see "Module documentation strings" on page 142). Docstrings most often span multiple physical lines, so you normally specify them in triple-quoted string literal form. For example:

```
def sum_args(*numbers):
    '''Accept arbitrary numerical arguments and return their sum.
    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 in any programming language, but their applicability is wider, since they remain available at runtime. 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" on page 454) makes it easy to check that sample code present in docstrings is accurate and correct.

To make your docstrings as useful as possible, you should 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 5. However, the ability to associate arbitrary attributes with a function can sometimes come in handy.

## The return Statement

The `return` statement in Python is allowed 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 executing `return None`).

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

Note that just *mentioning* a function (or other callable object) does *not* call it. To *call* a function (or other object) without arguments, you must use `()` after the function's name.

#### The semantics of argument passing

In traditional terms, all argument passing in Python is *by value*. 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." 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 = 77
b = [99]
f(a, b)
print a, b                # prints: 77 [99, 42]
```

The `print` statement 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, the `print` statement 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.

In a function call, zero or more positional arguments may be followed by zero or more *named arguments*, each with the following syntax:

*identifier=expression*

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. Most built-in functions do not accept named arguments, you must call such functions with positional arguments only. However, all normal functions coded in Python accept named as well as positional arguments, so you may call them in different ways.

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 // divisor
print divide(12, 94)                        # prints: 7
print 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+end
print f('tini', end='')                    # prints: inittini
```

Thanks to named argument `end=''`, the caller can specify a value, the empty string `''`, for `f`'s third parameter, `end`, and still let `f`'s second parameter, `begin`, use its default value, the string `'init'`.

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 described earlier in "Parameters" on page 71. 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, in this case, `print sum(d.values( ))` would be simpler and more direct!)

However, you may also pass arguments of the form *`*seq`* or *`**dct`* when calling a function that does not use the corresponding forms in its parameters. 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 names as its keys; otherwise, the call operation raises an exception.

## Namespaces

A function's parameters, plus any variables 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" on page 140. 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 rebind some global variables, the first statement of the function 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 rebind. For example, the function `counter` that we saw in "Other attributes of function objects" on page 73 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 often inelegant and unadvisable. As I mentioned earlier, when you want to group together some state and some behavior, the object-oriented mechanisms covered in Chapter 5 are usually best.

Don't use `global` if the function body just *uses* a global variable (including mutating the object bound to that variable if 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 will cause 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 explicitly pass that value as one of the function's arguments. If necessary, the argument's value can be bound when the nested function is defined 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 a tiny advantage: the computation of *a+b+c* happens only once, while `percent2`'s inner function `pc` repeats the computation three times. However, if the outer function rebinds its local variables between calls to the nested function, repeating the computation can be necessary. It's therefore advisable to be aware of both approaches, and choose the most 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 an exception to the general rule that the object-oriented mechanisms covered in Chapter 5 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.

## 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 a 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 `filter` on page 161):

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

As an alternative, you can always use a local `def` statement that gives the function object a name. You can then use this name as the argument or return value. Here's the same `filter` example using a local `def` statement:

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

While `lambda` can occasionally be useful, many Python users prefer `def`, which is more general, and may make your code more readable if you choose a reasonable 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*. When you call a generator, the function body does not execute. Instead, calling the generator returns a special iterator object 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 the `next` method of this iterator object is called, the function body executes up to the next `yield` statement, which takes the form:

```
yield expression
```

When a `yield` statement executes, the function execution is "frozen," with current point of execution and local variables intact, and the expression following `yield` is returned as the result of the `next` method. When `next` is called again, execution of the function body resumes where it left off, again up to the next `yield` statement. If the function body ends, or executes a `return` statement, the iterator raises a `StopIteration` exception to indicate that the iteration is finished. `return` statements in a generator cannot contain expressions.

A generator is 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:

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

Here is a generator that works somewhat like the built-in `xrange` function, but returns a sequence of floating-point values instead of a sequence of integers:

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

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

Generators are more flexible than functions that returns lists. A generator may build an *unbounded* iterator, meaning one that returns an infinite stream of results (to use only in loops that terminate by other means, e.g., via a `break` statement). Further, a generator-built 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 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:

```
resulting_list = list(G(arguments))
```

#### Generator expressions

Python 2.4 introduces 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" on page 67) except that a genexp is enclosed in 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, in any modern Python, you can code `sum([x*x for x in xrange(10)])`; in Python 2.4, you can express this functionality even better, coding it as `sum(x*x for x in xrange(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 in Python 2.5

In Python 2.5, generators are further enhanced, with the possibility of receiving a value (or an exception) back from the caller as each `yield` executes. These advanced features allow generators in 2.5 to implement full-fledged co-routines, as explained at *http://www.python.org/peps/pep-0342.html*. The main change is that, in 2.5, `yield` is not a statement, but an expression, so it has a value. When a generator is resumed by calling its method `next`, 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 `g.next()`, the caller calls `g.send(x)` (calling `g.send(None)` is just like calling `g.next()`). Also, a bare `yield` without arguments, in Python 2.5, becomes legal, and equivalent to `yield None`.

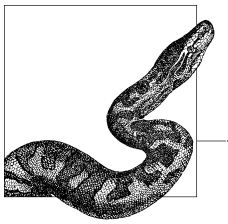
Other Python 2.5 enhancements to generators have to do with exceptions, and are covered in "Generator enhancements" on page 126.

## 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 `MaximumRecursionDepthExceeded` exception (covered in "Standard Exception Classes" on page 130) 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 `setrecursionlimit` on page 171.

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, be wary of trying to fix a program that is getting `MaximumRecursionDepthExceeded` exceptions by raising the recursion limit too high with `setrecursionlimit`. Most often, you'd be better advised to look for ways to remove the recursion or, more specifically, 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.



# 5

# Object-Oriented Python

Python is an object-oriented programming language. Unlike some other object-oriented languages, Python doesn't force you to use the object-oriented paradigm exclusively. Python also supports procedural programming with modules and functions, so you can select the most suitable programming paradigm for each part of your program. Generally, the object-oriented paradigm is suitable when you want to group state (data) and behavior (code) together in handy packets of functionality. It's also useful when you want to use some of Python's object-oriented mechanisms covered in this chapter, such as inheritance or special methods. The procedural paradigm, based on modules and functions, may be simpler, and thus more suitable when you don't need any of the benefits of object-oriented programming. With Python, you can mix and match the two paradigms.

Python today is in transition between two slightly different object models. This chapter mainly describes the *new-style*, or *new object model*, which is simpler, more regular, more powerful, and the one I recommend you *always* use; whenever I speak of classes or instances, without explicitly specifying otherwise, I mean new-style classes or instances. However, for backward compatibility, the default object model in all Python 2.*x* versions, for every value of *x*, is the *legacy object model*, also known as the *classic* or *old-style* object model; the new-style object model will become the default (and the legacy one will disappear) in a few years, when Python 3.0 comes out. Therefore, in each section, after describing how the new-style object model works, this chapter covers the small differences between the new and legacy object models, and discusses how to use both object models with Python 2.*x*. Finally, the chapter covers *special methods*, in "Special Methods" on page 104, and then two advanced concepts known as *decorators*, in "Decorators" on page 115, and *metaclasses*, in "Metaclasses" on page 116.

# Classes and Instances

If you're already familiar with object-oriented programming in other languages such as C++ or Java, then you probably have a good intuitive grasp of classes and instances: a *class* is a user-defined type, which you can *instantiate* to obtain *instances*, meaning objects of that type. Python supports these concepts through its class and instance objects.

## Python Classes

A *class* is a Python object with several characteristics:

- You can call a class object as if it were a function. The call returns another object, known as an *instance* of the class; the class is also known as the *type* of the instance.
- A class has arbitrarily named attributes that you can bind and reference.
- The values of class attributes can be *descriptors* (including functions), covered in "Descriptors" on page 85, or normal data objects.
- Class attributes bound to functions are also known as *methods* of the class.
- A method can have a special Python-defined name with two leading and two trailing underscores. Python implicitly invokes such *special methods*, if a class supplies them, when various kinds of operations take place on instances of that class.
- A class can *inherit* from other classes, meaning it delegates to other class objects the lookup of attributes that are not found in the class itself.

An instance of a class is a Python object with arbitrarily named attributes that you can bind and reference. An instance object implicitly delegates to its class the lookup of attributes not found in the instance itself. The class, in turn, may delegate the lookup to the classes from which it inherits, if any.

In Python, classes are objects (values) and are handled like other objects. Thus, you can pass a class as an argument in a call to a function. Similarly, a function can return a class as the result of a call. A class, just like any other object, can be bound to a variable (local or global), an item in a container, or an attribute of an object. Classes can also be keys into a dictionary. The fact that classes are ordinary objects in Python is often expressed by saying that classes are *first-class* objects.

## The class Statement

The `class` statement is the most common way to create a class object. `class` is a single-clause compound statement with the following syntax:

```
class classname(base-classes):
    statement(s)
```

*`classname`* is an identifier. It is a variable that gets bound (or rebound) to the class object after the `class` statement finishes executing.

*base-classes* is a comma-delimited series of expressions whose values must be class objects. These classes are known by different names in different programming languages; you can, depending on your choice, call them the *bases*, *superclasses*, or *parents* of the class being created. The class being created can be said to *inherit* from, *derive* from, *extend*, or *subclass* its base classes, depending on what programming language you are familiar with. This class is also known as a *direct subclass* or *descendant* of its base classes.

Syntactically, *base-classes* is optional: to indicate that you're creating a class without bases, you can omit *base-classes* (and the parentheses around it), placing the colon right after the classname (in Python 2.5, you may also use empty parentheses between the classname and the colon, with the same meaning). However, a class without bases, for reasons of backward compatibility, is an old-style one (unless you define the `__metaclass__` attribute, covered in "How Python Determines a Class's Metaclass" on page 117). To create a new-style class C without any "true" bases, code `class C(object):`; since every type subclasses the built-in object, specifying *object* as the value of base-classes just means that class C is new-style rather than old-style. If your class has ancestors that are all old-style and does not define the `__metaclass__` attribute, then your class is old-style; otherwise, a class with bases is always new-style (even if some bases are new-style and some are old-style).

The subclass relationship between classes is transitive: if *C1* subclasses *C2*, and *C2* subclasses *C3*, then *C1* subclasses *C3*. Built-in function `issubclass(C1, C2)` accepts two arguments that are class objects: it returns `True` if *C1* subclasses *C2*; otherwise, it returns `False`. Any class is considered a subclass of itself; therefore, `issubclass(C, C)` returns `True` for any class *C*. The way in which the base classes of a class affect the functionality of the class is covered in "Inheritance" on page 94.

The nonempty sequence of statements that follows the `class` statement is known as the *class body*. A class body executes immediately as part of the `class` statement's execution. Until the body finishes executing, the new class object does not yet exist and the *classname* identifier is not yet bound (or rebound). "How a Metaclass Creates a Class" on page 118 provides more details about what happens when a `class` statement executes.

Finally, note that the `class` statement does not immediately create any instance of the new class but rather defines the set of attributes that will be shared by all instances when you later create instances by calling the class.

## The Class Body

The body of a class is where you normally specify the attributes of the class; these attributes can be descriptor objects (including functions) or normal data objects of any type (an attribute of a class can also be another class—so, for example, you can have a `class` statement "nested" inside another `class` statement).

### Attributes of class objects

You normally specify an attribute of a class object by binding a value to an identifier within the class body. For example:

```
class C1(object):
    x = 23
print C1.x                              # prints: 23
```

Class object `C1` has an attribute named `x`, bound to the value `23`, and `C1.x` refers to that attribute.

You can also bind or unbind class attributes outside the class body. For example:

```
class C2(object): pass
C2.x = 23
print C2.x                              # prints: 23
```

However, your program is more readable if you bind, and thus create, class attributes with statements inside the class body. Any class attributes are implicitly shared by all instances of the class when those instances are created, as we'll discuss shortly.

The `class` statement implicitly sets some class attributes. Attribute `__name__` is the *`classname`* identifier string used in the `class` statement. Attribute `__bases__` is the tuple of class objects given as the base classes in the `class` statement. For example, using the class `C1` we just created:

```
print C1.__name__, C1.__bases__         # prints: C1, (<type 'object'>,)
```

A class also has an attribute `__dict__`, which is the dictionary object that the class uses to hold all of its other attributes. For any class object *`C`*, any object *`x`*, and any identifier *`S`* (except `__name__`, `__bases__`, and `__dict__`), *`C.S=x`* is equivalent to *`C`*`.__dict__['S']=`*`x`*. For example, again referring to the class `C1` we just created:

```
C1.y = 45
C1.__dict__['z'] = 67
print C1.x, C1.y, C1.z                  # prints: 23, 45, 67
```

There is no difference between class attributes created in the class body, outside the body by assigning an attribute, or outside the body by explicitly binding an entry in *`C`*`.__dict__`. (In Python 2.5, assignment to entries in the `__dict__` of a new-style class raises an exception.)

In statements that are directly in a class's body, references to attributes of the class must use a simple name, not a fully qualified name. For example:

```
class C3(object):
    x = 23
    y = x + 22                          # must use just x, not C3.x
```

However, in statements that are in methods defined in a class body, references to attributes of the class must use a fully qualified name, not a simple name. For example:

```
class C4(object):
    x = 23
    def amethod(self):
        print C4.x                      # must use C4.x, not just x
```

Note that attribute references (i.e., an expression like `C.s`) have semantics richer than those of attribute bindings. I cover these references in detail in "Attribute Reference Basics" on page 89.

#### Function definitions in a class body

Most class bodies include `def` statements, since functions (called methods in this context) are important attributes for most class objects. A `def` statement in a class body obeys the rules presented in "Functions" on page 70. In addition, a method defined in a class body always has a mandatory first parameter, conventionally named `self`, that refers to the instance on which you call the method. The `self` parameter plays a special role in method calls, as covered in "Bound and Unbound Methods" on page 91.

Here's an example of a class that includes a method definition:

```
class C5(object):
    def hello(self):
        print "Hello"
```

A class can define a variety of special methods (methods with names that have two leading and two trailing underscores) relating to specific operations on its instances. I discuss special methods in detail in "Special Methods" on page 104.

#### Class-private variables

When a statement in a class body (or in a method in the body) uses an identifier starting with two underscores (but not ending with underscores), such as `__ident`, the Python compiler implicitly changes the identifier into `_classname__ident`, where *classname* is the name of the class. This lets a class use "private" names for attributes, methods, global variables, and other purposes, reducing the risk of accidentally duplicating names used elsewhere.

By convention, all identifiers starting with a single underscore are meant to be private to the scope that binds them, whether that scope is or isn't a class. The Python compiler does not enforce this privacy convention; it's up to Python programmers to respect it.

#### Class documentation strings

If the first statement in the class body is a string literal, the compiler binds that string as the documentation string attribute for the class. This attribute is named `__doc__` and is known as the *docstring* of the class. See "Docstrings" on page 72 for more information on docstrings.

## Descriptors

A *descriptor* is any new-style object whose class supplies a special method named `__get__`. Descriptors that are class attributes control the semantics of accessing and setting attributes on instances of that class. Roughly speaking, when you access an instance attribute, Python obtains the attribute's value by calling

__get__ on the corresponding descriptor, if any. For more details, see "Attribute Reference Basics" on page 89.

#### Overriding and nonoverriding descriptors

If a descriptor's class also supplies a special method named __set__, then the descriptor is known as an *overriding descriptor* (or, by an older and slightly confusing terminology, a *data descriptor*); if the descriptor's class supplies only __get__, and not __set__, then the descriptor is known as a *nonoverriding* (or *nondata*) *descriptor*. For example, the class of function objects supplies __get__, but not __set__; therefore, function objects are nonoverriding descriptors. Roughly speaking, when you assign a value to an instance attribute with a corresponding descriptor that is overriding, Python sets the attribute value by calling __set__ on the descriptor. For more details, see "Attributes of instance objects" on page 87.

Old-style classes can have descriptors, but descriptors in old-style classes always work as if they were nonoverriding ones (their __set__ method, if any, is ignored).

## Instances

To create an instance of a class, call the class object as if it were a function. Each call returns a new instance whose type is that class:

```
anInstance = C5()
```

You can call built-in function isinstance(*I*, *C*) with a class object as argument *C*. isinstance returns True if object *I* is an instance of class *C* or any subclass of *C*. Otherwise, isinstance returns False.

#### __init__

When a class defines or inherits a method named __init__, calling the class object implicitly executes __init__ on the new instance to perform any needed instance-specific initialization. Arguments passed in the call must correspond to the parameters of __init__, except for parameter self. For example, consider the following class:

```
class C6(object):
    def __init__(self, n):
        self.x = n
```

Here's how you can create an instance of the C6 class:

```
anotherInstance = C6(42)
```

As shown in the C6 class, the __init__ method typically contains statements that bind instance attributes. An __init__ method must not return a value; otherwise, Python raises a TypeError exception.

The main purpose of __init__ is to bind, and thus create, the attributes of a newly created instance. You may also bind or unbind instance attributes outside __init__, as you'll see shortly. However, your code will be more readable if you

initially bind all attributes of a class instance with statements in the `__init__` method.

When `__init__` is absent, you must call the class without arguments, and the newly generated instance has no instance-specific attributes.

#### Attributes of instance objects

Once you have created an instance, you can access its attributes (data and methods) using the dot (`.`) operator. For example:

```
anInstance.hello( )                    # prints: Hello
print anotherInstance.x                # prints: 42
```

Attribute references such as these have fairly rich semantics in Python and are covered in detail in "Attribute Reference Basics" on page 89.

You can give an instance object an arbitrary attribute by binding a value to an attribute reference. For example:

```
class C7(object): pass
z = C7( )
z.x = 23
print z.x                                  # prints: 23
```

Instance object `z` now has an attribute named `x`, bound to the value `23`, and `z.x` refers to that attribute. Note that the `__setattr__` special method, if present, intercepts every attempt to bind an attribute. (`__setattr__` is covered in `__setattr__` on page 108.) Moreover, if you attempt to bind, on a new-style instance, an attribute whose name corresponds to an overriding descriptor in the instance's class, the descriptor's `__set__` method intercepts the attempt. In this case, the statement `z.x=23` executes `type(z).x.__set__(z, 23)` (old-style instances ignore the overriding nature of descriptors found in their classes, i.e., they never call their `__set__` methods).

Creating an instance implicitly sets two instance attributes. For any instance *z*, *z*`.__class__` is the class object to which *z* belongs, and *z*`.__dict__` is the dictionary that *z* uses to hold its other attributes. For example, for the instance `z` we just created:

```
print z.__class__.__name__, z.__dict__     # prints: C7, {'x':23}
```

You may rebind (but not unbind) either or both of these attributes, but this is rarely necessary. A new-style instance's `__class__` may be rebound only to a new-style class, and a legacy instance's `__class__` may be rebound only to a legacy class.

For any instance *z*, any object *x*, and any identifier *S* (except `__class__` and `__dict__`), *z*`.`*S*`=`*x* is equivalent to *z*`.__dict__['`*S*`']=`*x* (unless a `__setattr__` special method, or an overriding descriptor's `__set__` special method, intercept the binding attempt). For example, again referring to the *z* we just created:

```
z.y = 45
z.__dict__['z'] = 67
print z.x, z.y, z.z                        # prints: 23, 45, 67
```

There is no difference between instance attributes created in `__init__` by assigning to attributes or by explicitly binding an entry in `z.__dict__`.

#### The factory-function idiom

A common task is to create instances of different classes depending on some condition, or to avoid creating a new instance if an existing one is available for reuse. A common misconception is that such needs might be met by having `__init__` return a particular object, but such an approach is absolutely unfeasible: Python raises an exception when `__init__` returns any value other than `None`. The best way to implement flexible object creation is by using an ordinary function rather than calling the class object directly. A function used in this role is known as a *factory function*.

Calling a factory function is a flexible approach: a function may return an existing reusable instance, or create a new instance by calling whatever class is appropriate. Say you have two almost interchangeable classes (`SpecialCase` and `NormalCase`) and want to flexibly generate instances of either one of them, depending on an argument. The following `appropriateCase` factory function allows you to do just that (the role of the `self` parameter is covered in "Bound and Unbound Methods" on page 91):

```
class SpecialCase(object):
    def amethod(self): print "special"
class NormalCase(object):
    def amethod(self): print "normal"
def appropriateCase(isnormal=True):
    if isnormal: return NormalCase( )
    else: return SpecialCase( )
aninstance = appropriateCase(isnormal=False)
aninstance.amethod( )                    # prints "special", as desired
```

#### `__new__`

Each new-style class has (or inherits) a static method named `__new__` (static methods are covered in "Static methods" on page 99). When you call `C(*args,**kwds)` to create a new instance of class `C`, Python first calls `C.__new__(C,*args,**kwds)`. Python uses `__new__`'s return value `x` as the newly created instance. Then, Python calls `C.__init__(x,*args,**kwds)`, but only when `x` is indeed an instance of `C` or any of its subclasses (otherwise, `x`'s state remains as `__new__` had left it). Thus, for example, the statement `x=C(23)` is equivalent to:

```
x = C.__new__(C, 23)
if isinstance(x, C): type(x).__init__(x, 23)
```

`object.__new__` creates a new, uninitialized instance of the class it receives as its first argument. It ignores other arguments if that class has an `__init__` method, but it raises an exception if it receives other arguments beyond the first, and the class that's the first argument does not have an `__init__` method. When you override `__new__` within a class body, you do not need to add `__new__=staticmethod(__new__)`, as you normally would: Python recognizes the name `__new__` and treats it specially in this context. In those rare cases in which you rebind `C.__new__` later, outside the body of class `C`, you do need to use `C.__new__=staticmethod(whatever)`.

__new__ has most of the flexibility of a factory function, as covered in "The factory-function idiom" on page 88. __new__ may choose to return an existing instance or make a new one, as appropriate. When __new__ does need to create a new instance, it most often delegates creation by calling object.__new__ or the __new__ method of another superclass of C. The following example shows how to override static method __new__ in order to implement a version of the Singleton design pattern:

```
class Singleton(object):
    _singletons = {}
    def __new__(cls, *args, **kwds):
        if cls not in cls._singletons:
            cls._singletons[cls] = super(Singleton, cls).__new__(cls)
        return cls._singletons[cls]
```

(Built-in super is covered in "Cooperative superclass method calling" on page 97.) Any subclass of Singleton (that does not further override __new__) has exactly one instance. If the subclass defines an __init__ method, the subclass must ensure its __init__ is safe when called repeatedly (at each creation request) on the one and only class instance.

Old-style classes do not have a __new__ method.

## Attribute Reference Basics

An *attribute reference* is an expression of the form *x.name*, where *x* is any expression and *name* is an identifier called the *attribute name*. Many kinds of Python objects have attributes, but an attribute reference has special rich semantics when *x* refers to a class or instance. Remember that methods are attributes too, so everything I say about attributes in general also applies to attributes that are callable (i.e., methods).



Say that x is an instance of class C, which inherits from base class B. Both classes and the instance have several attributes (data and methods), as follows:

```
class B(object):
    a = 23
    b = 45
    def f(self): print "method f in class B"
    def g(self): print "method g in class B"
class C(B):
    b = 67
    c = 89
    d = 123
    def g(self): print "method g in class C"
    def h(self): print "method h in class C"
x = C()
x.d = 77
x.e = 88
```

A few attribute names are special. For example, C.__name__ is the string 'C' and the class name. C.__bases__ is the tuple (B,), the tuple of C's base classes. x.__class__ is the class C, the class to which x belongs. When you refer to an attribute with one of these special names, the attribute reference looks directly into a dedicated slot in the class or instance object and fetches the value it finds there. You cannot unbind

these attributes. Rebinding them is allowed, so you can change the name or base classes of a class, or the class of an instance, on the fly, but this advanced technique is rarely necessary.

Both class `C` and instance `x` each have one other special attribute: a dictionary named `__dict__`. All other attributes of a class or instance, except for the few special ones, are held as items in the `__dict__` attribute of the class or instance.

#### Getting an attribute from a class

When you use the syntax *`C.name`* to refer to an attribute on a class object *`C`*, the lookup proceeds in two steps:

1. When '*`name`*' is a key in *`C`*`.__dict__`, *`C.name`* fetches the value *`v`* from *`C`*`.__dict__['`*`name`*`']`. Then, if *`v`* is a descriptor (i.e., `type(`*`v`*`)` supplies a method named `__get__`), the value of *`C.name`* is the result of calling `type(`*`v`*`).__get__(v, None, `*`C`*`)`. Otherwise, the value of *`C.name`* is *`v`*.
2. Otherwise, *`C.name`* delegates the lookup to *`C`*'s base classes, meaning it loops on *`C`*'s ancestor classes and tries the *`name`* lookup on each (in "method resolution order," as covered in "Method resolution order" on page 94).

#### Getting an attribute from an instance

When you use the syntax *`x.name`* to refer to an attribute of instance *`x`* of class *`C`*, the lookup proceeds in three steps:

1. When '*`name`*' is found in *`C`* (or in one of *`C`*'s ancestor classes) as the name of an overriding descriptor *`v`* (i.e., `type(`*`v`*`)` supplies methods `__get__` and `__set__`), the value of *`C.name`* is the result of calling `type(`*`v`*`).__get__(v, `*`x`*`, `*`C`*`)`. (This step doesn't apply to old-style instances).
2. Otherwise, when '*`name`*' is a key in *`x`*`.__dict__`, *`x.name`* fetches and returns the value at *`x`*`.__dict__['`*`name`*`']`.
3. Otherwise, *`x.name`* delegates the lookup to *`x`*'s class (according to the same two-step lookup used for *`C.name`*, as just detailed). If a descriptor *`v`* is found, the overall result of the attribute lookup is, again, `type(`*`v`*`).__get__(v, `*`x`*`, `*`C`*`)`; if a nondescriptor value *`v`* is found, the overall result of the attribute lookup is *`v`*.

When these lookup steps do not find an attribute, Python raises an `AttributeError` exception. However, for lookups of *`x.name`*, if *`C`* defines or inherits special method `__getattr__`, Python calls *`C`*`.__getattr__(`*`x`*`,'`*`name`*`')` rather than raising the exception (it's then up to `__getattr__` to either return a suitable value or raise the appropriate exception, normally `AttributeError`).

Consider the following attribute references:

```
print x.e, x.d, x.c, x.b, x.a           # prints: 88, 77, 89, 67, 23
```

`x.e` and `x.d` succeed in step 2 of the instance lookup process, since no descriptors are involved, and `'e'` and `'d'` are both keys in `x.__dict__`. Therefore, the lookups go no further, but rather return `88` and `77`. The other three references must proceed to step 3 of the instance process and look in `x.__class__` (i.e., `C`). `x.c` and `x.b` succeed in step 1 of the class lookup process, since `'c'` and `'b'` are both keys in `C.__dict__`. Therefore, the lookups go no further but rather return `89` and `67`.

`x.a` gets all the way to step 2 of the class process, looking in `C.__bases__[0]` (i.e., `B`). `'a'` is a key in `B.__dict__`; therefore, `x.a` finally succeeds and returns `23`.

#### Setting an attribute

Note that the attribute lookup steps happen in this way only when you refer to an attribute, not when you bind an attribute. When you bind (on either a class or an instance) an attribute whose name is not special (unless a `__setattr__` method, or the `__set__` method of an overriding descriptor, intercepts the binding of an instance attribute), you affect only the `__dict__` entry for the attribute (in the class or instance, respectively). In other words, in the case of attribute binding, there is no lookup procedure involved, except for the check for overriding descriptors.

## Bound and Unbound Methods

Method `__get__` of a function object returns an *unbound method object* or a *bound method object* that wraps the function. The key difference between unbound and bound methods is that an unbound method is not associated with a particular instance while a bound method is.

In the code in the previous section, attributes `f`, `g`, and `h` are functions; therefore, an attribute reference to any one of them returns a method object that wraps the respective function. Consider the following:

```
print x.h, x.g, x.f, C.h, C.g, C.f
```

This statement outputs three bound methods represented by strings like:

```
<bound method C.h of <__main__.C object at 0x8156d5c>>
```

and then three unbound ones represented by strings like:

```
<unbound method C.h>
```

We get bound methods when the attribute reference is on instance `x`, and unbound methods when the attribute reference is on class `C`.

Because a bound method is already associated with a specific instance, you call the method as follows:

```
x.h( )                      # prints: method h in class C
```

The key thing to notice here is that you don't pass the method's first argument, `self`, by the usual argument-passing syntax. Rather, a bound method of instance `x` implicitly binds the `self` parameter to object `x`. Thus, the body of the method can access the instance's attributes as attributes of `self`, even though we don't pass an explicit argument to the method.

An unbound method, however, is not associated with a specific instance, so you must specify an appropriate instance as the first argument when you invoke an unbound method. For example:

```
C.h(x)                      # prints: method h in class C
```

You call unbound methods far less frequently than bound methods. The main use for unbound methods is for accessing overridden methods, as discussed in

"Inheritance" on page 94; moreover, even for that task, it's generally better to use the `super` built-in covered in "Cooperative superclass method calling" on page 97.

#### Unbound method details

As we've just discussed, when an attribute reference on a class refers to a function, a reference to that attribute returns an unbound method that wraps the function. An unbound method has three attributes in addition to those of the function object it wraps: `im_class` is the class object supplying the method, `im_func` is the wrapped function, and `im_self` is always `None`. These attributes are all read-only, meaning that trying to rebind or unbind any of them raises an exception.

You can call an unbound method just as you would call its `im_func` function, but the first argument in any call must be an instance of `im_class` or a descendant. In other words, a call to an unbound method must have at least one argument, which corresponds to the wrapped function's first formal parameter (conventionally named `self`).

#### Bound method details

When an attribute reference on an instance, in the course of the lookup, finds a function object that's an attribute in the instance's class, the lookup calls the function's `__get__` method to obtain the attribute's value. The call, in this case, creates and returns a *bound method* that wraps the function.

Note that when the attribute reference's lookup finds a function object in *x*`.__dict__`, the attribute reference operation does *not* create a bound method because in such cases the function is not treated as a descriptor, and the function's `__get__` method does not get called; rather, the function object itself is the attribute's value. Similarly, no bound method is created for callables that are not ordinary functions, such as built-in (as opposed to Python-coded) functions, since they are not descriptors.

A bound method is similar to an unbound method in that it has three read-only attributes in addition to those of the function object it wraps. Like in an unbound method, `im_class` is the class object that supplies the method, and `im_func` is the wrapped function. However, in a bound method object, attribute `im_self` refers to *x*, the instance from which the method was obtained.

A bound method is used like its `im_func` function, but calls to a bound method do not explicitly supply an argument corresponding to the first formal parameter (conventionally named `self`). When you call a bound method, the bound method passes `im_self` as the first argument to `im_func` before other arguments (if any) given at the point of call.

Let's follow in excruciating low-level detail the conceptual steps involved in a method call with the normal syntax *x*`.`*name*`(`*arg*`)`. In the following context:

```
def f(a, b): ...              # a function f with two arguments
```

```
class C(object):
    name = f
x = C()
```

*x* is an instance object of class `C`, *name* is an identifier that names a method of *x*'s (an attribute of `C` whose value is a function, in this case function `f`), and *arg* is any expression. Python first checks if '*name*' is the attribute name in `C` of an overriding descriptor, but it isn't—functions are descriptors, because their class defines method `__get__`, but not overriding ones, because their class does not define method `__set__`. Python next checks if '*name*' is a key in *x*`.__dict__`, but it isn't. So Python finds *name* in `C` (everything would work in just the same way if *name* was found, by inheritance, in one of `C`'s `__bases__`). Python notices that the attribute's value, function object *f*, is a descriptor. Therefore, Python calls *f*`.__get__(`*x*`, C)`, which creates a bound method object with `im_func` set to `f`, `im_class` set to `C`, and `im_self` set to *x*. Then Python calls this bound method object, with *arg* as the only actual argument. The bound method inserts `im_self` (i.e., *x*) as the first actual argument, and *arg* becomes the second one, in a call to the bound method's `im_func` (i.e., function `f`). The overall effect is just like calling:

```
x.__class__.__dict__['name'](x, arg)
```

When a bound method's function body executes, it has no special namespace relationship to either its `self` object or any class. Variables referenced are local or global, just as for any other function, as covered in "Namespaces" on page 76. Variables do not implicitly indicate attributes in `self`, nor do they indicate attributes in any class object. When the method needs to refer to, bind, or unbind an attribute of its `self` object, it does so by standard attribute-reference syntax (e.g., `self.`*name*). The lack of implicit scoping may take some getting used to (since Python differs in this respect from many other object-oriented languages), but it results in clarity, simplicity, and the removal of potential ambiguities.



Bound method objects are first-class objects, and you can use them wherever you can use a callable object. Since a bound method holds references to the function it wraps, and to the `self` object on which it executes, it's a powerful and flexible alternative to a closure (covered in "Nested functions and nested scopes" on page 77). An instance object whose class supplies special method `__call__` (covered in `__call__` on page 105) offers another viable alternative. Each of these constructs lets you bundle some behavior (code) and some state (data) into a single callable object. Closures are simplest, but limited in their applicability. Here's the closure from "Nested functions and nested scopes" on page 77:

```
def make_adder_as_closure(augend):
    def add(addend, _augend=augend): return addend+_augend
    return add
```

Bound methods and callable instances are richer and more flexible than closures. Here's how to implement the same functionality with a bound method:

```
def make_adder_as_bound_method(augend):
    class Adder(object):
        def __init__(self, augend): self.augend = augend
        def add(self, addend): return addend+self.augend
    return Adder(augend).add
```

Here's how to implement it with a callable instance (an instance whose class supplies special method `__call__`):

```
def make_adder_as_callable_instance(augend):
    class Adder(object):
        def __init__(self, augend): self.augend = augend
        def __call__(self, addend): return addend+self.augend
    return Adder(augend)
```

From the viewpoint of the code that calls the functions, all of these factory functions are interchangeable, since all of them return callable objects that are polymorphic (i.e., usable in the same ways). In terms of implementation, the closure is simplest; the bound method and the callable instance use more flexible, general, and powerful mechanisms, but there is really no need for that extra power in this simple example.

## Inheritance

When you use an attribute reference *`C.name`* on a class object *`C`*, and *`'name'`* is not a key in `C.__dict__`, the lookup implicitly proceeds on each class object that is in `C.__bases__` in a specific order (which for historical reasons is known as the *method resolution order*, or MRO, even though it's used for all attributes, not just methods). *`C`*'s base classes may in turn have their own bases. The lookup checks direct and indirect ancestors, one by one, in MRO, stopping when *`'name'`* is found.

### Method resolution order

The lookup of an attribute name in a class essentially occurs by visiting ancestor classes in left-to-right, depth-first order. However, in the presence of multiple inheritance (which makes the inheritance graph a general Directed Acyclic Graph rather than specifically a tree), this simple approach might lead to some ancestor class being visited twice. In such cases, the resolution order is clarified by leaving in the lookup sequence only the *rightmost* occurrence of any given class. This last, crucial simplification is not part of the specifications for the legacy object model, making multiple inheritance hard to use correctly and effectively within that object model. The new-style object model is vastly superior in this regard.

The problem with purely left-right, depth-first search, in situations of multiple inheritance, can be easily demonstrated with an example based on old-style classes:

```
class Base1:
    def amethod(self): print "Base1"
class Base2(Base1): pass
class Base3(Base1):
    def amethod(self): print "Base3"
class Derived(Base2, Base3): pass
aninstance = Derived( )
aninstance.amethod( )                   # prints: "Base1"
```

In this case, the lookup for `amethod` starts in `Derived`. When it isn't found there, lookup proceeds to `Base2`. Since the attribute isn't found in `Base2`, the legacy-style lookup then proceeds to `Base2`'s ancestor, `Base1`, where the attribute is found. Therefore, the legacy-style lookup stops at this point and never considers `Base3`, where it would also find an attribute with the same name. The new-style MRO solves this problem by removing the leftmost occurrence of `Base1` from the search so that the occurrence of `amethod` in `Base3` is found instead.

Figure 5-1 shows the legacy and new-style MROs for the case of this kind of "diamond-shaped" inheritance graph.

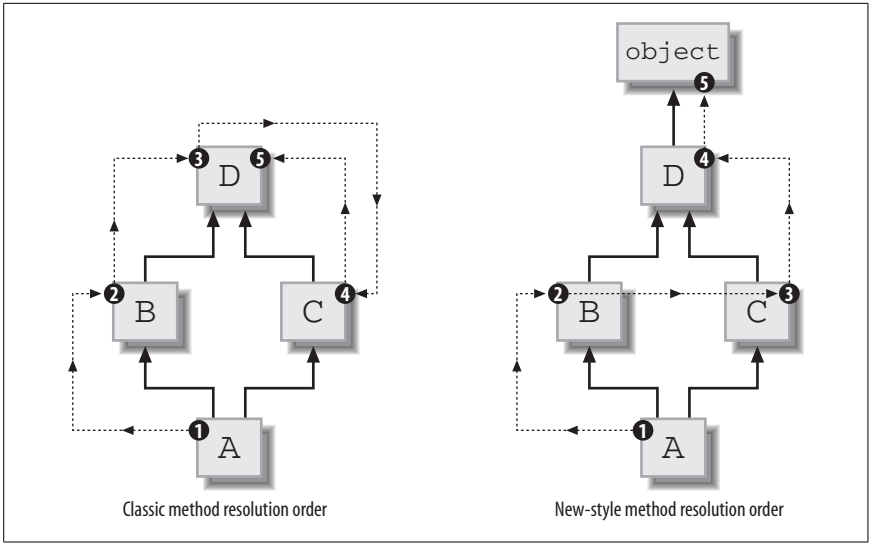



*Figure 5-1. Legacy and new-style MRO*

Each new-style class and built-in type has a special read-only class attribute called `__mro__`, which is the tuple of types used for method resolution, in order. You can reference `__mro__` only on classes, not on instances, and, since `__mro__` is a read-only attribute, you cannot rebind or unbind it. For a detailed and highly technical explanation of all aspects of Python's MRO, you may want to study a paper by Michele Simionato, "The Python 2.3 Method Resolution Order," at *http://www.python.org/2.3/mro.html*.

### Overriding attributes

As we've just seen, the search for an attribute proceeds along the MRO (typically up the inheritance tree) and stops as soon as the attribute is found. Descendant classes are always examined before their ancestors so that, when a subclass defines an attribute with the same name as one in a superclass, the search finds the definition in the subclass and stops there. This is known as the subclass *overriding* the definition in the superclass. Consider the following:

```
class B(object):
    a = 23
    b = 45
    def f(self): print "method f in class B"
    def g(self): print "method g in class B"
class C(B):
    b = 67
    c = 89
    d = 123
    def g(self): print "method g in class C"
    def h(self): print "method h in class C"
```

In this code, class `C` overrides attributes `b` and `g` of its superclass `B`. Note that, unlike in some other languages, in Python you may override data attributes just as easily as callable attributes (methods).

#### Delegating to superclass methods

When a subclass `C` overrides a method `f` of its superclass `B`, the body of `C.f` often wants to delegate some part of its operation to the superclass's implementation of the method. This can sometimes be done using an unbound method, as follows:

```
class Base(object):
    def greet(self, name): print "Welcome ", name
class Sub(Base):
    def greet(self, name):
        print "Well Met and",
        Base.greet(self, name)
x = Sub( )
x.greet('Alex')
```

The delegation to the superclass, in the body of `Sub.greet`, uses an unbound method obtained by attribute reference `Base.greet` on the superclass, and therefore passes all attributes normally, including `self`. Delegating to a superclass implementation is the most frequent use of unbound methods.

One common use of delegation occurs with special method `__init__`. When Python creates an instance, the `__init__` methods of base classes are not automatically invoked, as they are in some other object-oriented languages. Thus, it is up to a subclass to perform the proper initialization by using delegation if necessary. For example:

```
class Base(object):
    def __init__(self):
        self.anattribute = 23
class Derived(Base):
    def __init__(self):
        Base.__init__(self)
        self.anotherattribute = 45
```

If the `__init__` method of class `Derived` didn't explicitly call that of class `Base`, instances of `Derived` would miss that portion of their initialization, and thus such instances would lack attribute `anattribute`.

### Cooperative superclass method calling

Calling the superclass's version of a method with unbound method syntax, however, is quite problematic in cases of multiple inheritance with diamond-shaped graphs. Consider the following definitions:

```
class A(object):
    def met(self):
        print 'A.met'
class B(A):
    def met(self):
        print 'B.met'
        A.met(self)
class C(A):
    def met(self):
        print 'C.met'
        A.met(self)
class D(B,C):
    def met(self):
        print 'D.met'
        B.met(self)
        C.met(self)
```

In this code, when we call `D().met()`, `A.met` ends up being called twice. How can we ensure that each ancestor's implementation of the method is called once, and only once? The solution is to use built-in type `super`. `super(`*`aclass, obj`*`)`, which returns a special superobject of object *`obj`*. When we look up an attribute (e.g., a method) in this superobject, the lookup begins *after* class *`aclass`* in *`obj`*'s MRO. We can therefore rewrite the previous code as:

```
class A(object):
    def met(self):
        print 'A.met'
class B(A):
    def met(self):
        print 'B.met'
        super(B,self).met()
class C(A):
    def met(self):
        print 'C.met'
        super(C,self).met()
class D(B,C):
    def met(self):
        print 'D.met'
        super(D,self).met()
```

Now, `D().met()` results in exactly one call to each class's version of `met`. If you get into the habit of always coding superclass calls with `super`, your classes will fit smoothly even in complicated inheritance structures. There are no ill effects whatsoever if the inheritance structure instead turns out to be simple, as long, of course, as you're only using the new-style object model, as I recommend.

The only situation in which you may prefer to use the rougher approach of calling a superclass method through the unbound-method technique is when the various classes have different and incompatible signatures for the same method—an unpleasant situation in many respects, but, if you do have to deal with it, the unbound-method technique may sometimes be the least of evils. Proper use of multiple inheritance will be seriously hampered—but then, even the most fundamental properties of OOP, such as polymorphism between base and subclass instances, are seriously impaired when corresponding methods have different and incompatible signatures.

#### "Deleting" class attributes

Inheritance and overriding provide a simple and effective way to add or modify class attributes (particularly methods) noninvasively (i.e., without modifying the class in which the attributes are defined) by adding or overriding the attributes in subclasses. However, inheritance does not offer a way to delete (hide) base classes' attributes noninvasively. If the subclass simply fails to define (override) an attribute, Python finds the base class's definition. If you need to perform such deletion, possibilities include:

- Override the method and raise an exception in the method's body.
- Eschew inheritance, hold the attributes elsewhere than in the subclass's `__dict__`, and define `__getattr__` for selective delegation.
- Use the new-style object model and override `__getattribute__` to similar effect.

The last of these techniques is demonstrated in "`__`getattribute`__`" on page 102.

## The Built-in object Type

The built-in `object` type is the ancestor of all built-in types and new-style classes. The `object` type defines some special methods (documented in "Special Methods" on page 104) that implement the default semantics of objects:

`__new__`
`__init__`
: You can create a direct instance of `object` by calling `object()` without any arguments. The call implicitly uses `object.__new__` and `object.__init__` to make and return an instance object without attributes (and without even a `__dict__` in which to hold attributes). Such an instance object may be useful as a "sentinel," guaranteed to compare unequal to any other distinct object.

`__delattr__`
`__getattribute__`
`__setattr__`
: By default, an object handles attribute references (as covered in "Attribute Reference Basics" on page 89) using these methods of `object`.

`__hash__`
`__repr__`
`__str__`
: Any object can be passed to functions `hash` and `repr` and to type `str`.

A subclass of `object` may override any of these methods and/or add others.

## Class-Level Methods

Python supplies two built-in nonoverriding descriptors types, which give a class two distinct kinds of "class-level methods."

### Static methods

A *static method* is a method that you can call on a class, or on any instance of the class, without the special behavior and constraints of ordinary methods, bound and unbound, with regard to the first parameter. A static method may have any signature; it might have no parameters, and the first parameter, if it does have any, plays no special role. You can think of a static method as an ordinary function that you're able to call normally, despite the fact that it happens to be bound to a class attribute. While it is never necessary to define static methods (you can always define a normal function instead), some programmers consider them to be an elegant alternative when a function's purpose is tightly bound to some specific class.

To build a static method, call built-in type `staticmethod` and bind its result to a class attribute. Like all binding of class attributes, this is normally done in the body of the class, but you may also choose to perform it elsewhere. The only argument to `staticmethod` is the function to invoke when Python calls the static method. The following example shows how to define and call a static method:

```
class AClass(object):
    def astatic(): print 'a static method'
    astatic = staticmethod(astatic)
anInstance = AClass()
AClass.astatic()                    # prints: a static method
anInstance.astatic()                # prints: a static method
```

This example uses the same name for the function passed to `staticmethod` and for the attribute bound to `staticmethod`'s result. This style is not mandatory, but it's a good idea, and I recommend you always use it. Python 2.4 offers a special, simplified syntax to support this style, covered in "Decorators" on page 115.

### Class methods

A *class method* is a method you can call on a class or on any instance of the class. Python binds the method's first parameter to the class on which you call the method, or the class of the instance on which you call the method; it does not bind it to the instance, as for normal bound methods. There is no equivalent of unbound methods for class methods. The first parameter of a class method is conventionally named `cls`. While it is never necessary to define class methods (you could always alternatively define a normal function that takes the class object as its first parameter), some programmers consider them to be an elegant alternative to such functions.

To build a class method, call built-in type `classmethod` and bind its result to a class attribute. Like all binding of class attributes, this is normally done in the

body of the class, but you may also choose to perform it elsewhere. The only argument to `classmethod` is the function to invoke when Python calls the class method. Here's how you can define and call a class method:

```
class ABase(object):
    def aclassmet(cls): print 'a class method for', cls.__name__
    aclassmet = classmethod(aclassmet)
class ADeriv(ABase): pass
bInstance = ABase( )
dInstance = ADeriv( )
ABase.aclassmet( )              # prints: a class method for ABase
bInstance.aclassmet( )          # prints: a class method for ABase
ADeriv.aclassmet( )             # prints: a class method for ADeriv
dInstance.aclassmet( )          # prints: a class method for ADeriv
```

This example uses the same name for the function passed to `classmethod` and for the attribute bound to `classmethod`'s result. This style is not mandatory, but it's a good idea, and I recommend that you always use it. Python 2.4 offers a special, simplified syntax to support this style, covered in "Decorators" on page 115.

## Properties

Python supplies a built-in overriding descriptor type, which you may use to give a class's instances *properties*.

A property is an instance attribute with special functionality. You reference, bind, or unbind the attribute with the normal syntax (e.g., `print` *`x`*`.prop`, *`x`*`.prop=23`, `del` *`x`*`.prop`). However, rather than following the usual semantics for attribute reference, binding, and unbinding, these accesses call on instance *`x`* the methods that you specify as arguments to the built-in type `property`. Here's how you define a read-only property:

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def getArea(self):
        return self.width * self.height
    area = property(getArea, doc='area of the rectangle')
```

Each instance *`r`* of class `Rectangle` has a synthetic read-only attribute *`r`*`.area`, computed on the fly in method *`r`*`.getArea( )` by multiplying the sides of the rectangle. The docstring `Rectangle.area.__doc__` is `'area of the rectangle'`. Attribute *`r`*`.area` is read-only (attempts to rebind or unbind it fail) because we specify only a `get` method in the call to `property`, no `set` or `del` methods.

Properties perform tasks similar to those of special methods `__getattr__`, `__setattr__`, and `__delattr__` (covered in "General-Purpose Special Methods" on page 104), but in a faster and simpler way. You build a property by calling built-in type `property` and binding its result to a class attribute. Like all binding of class attributes, this is normally done in the body of the class, but you may also choose to perform it elsewhere. Within the body of a class *`C`*, use the following syntax:

```
attrib = property(fget=None, fset=None, fdel=None, doc=None)
```

When *x* is an instance of *C* and you reference *x.attrib*, Python calls on *x* the method you passed as argument *fget* to the property constructor, without arguments. When you assign *x.attrib* = *value*, Python calls the method you passed as argument *fset*, with *value* as the only argument. When you execute `del` *x.attrib*, Python calls the method you passed as argument *fdel*, without arguments. Python uses the argument you passed as *doc* as the docstring of the attribute. All parameters to `property` are optional. When an argument is missing, the corresponding operation is forbidden (Python raises an exception when some code attempts that operation). For example, in the `Rectangle` example, we made property `area` read-only, because we passed an argument only for parameter *fget*, and not for parameters *fset* and *fdel*.

#### Why properties are important

The crucial importance of properties is that their existence makes it perfectly safe and indeed advisable for you to expose public data attributes as part of your class's public interface. If it ever becomes necessary, in future versions of your class or other classes that need to be polymorphic to it, to have some code executed when the attribute is referenced, rebound, or unbound, you know you will be able to change the plain attribute into a property and get the desired effect without any impact on any other code that uses your class (a.k.a. "client code"). This lets you avoid goofy idioms, such as *accessor* and *mutator* methods, required by OO languages that lack properties or equivalent machinery. For example, client code can simply use natural idioms such as:

```
someInstance.widgetCounter += 1
```

rather than being forced into contorted nests of accessors and mutators such as:

```
someInstance.setWidgetCounter(someInstance.getWidgetCounter( ) + 1)
```

If at any time you're tempted to code methods whose natural names are something like *getThis* or *setThat*, consider wrapping those methods into properties, for clarity.

#### Properties and inheritance

Properties are inherited normally, just like any other attribute. However, there's a little trap for the unwary: the methods called upon to access a property are those that are defined in the class in which the property itself is defined, without intrinsic use of further overriding that may happen in subclasses. For example:

```
class B(object):
  def f(self): return 23
  g = property(f)
class C(B):
  def f(self): return 42
c = C( )
print c.g                # prints 23, not 42
```

The access to property *c.g* calls *B.f*, not *C.f* as you might intuitively expect. The reason is quite simple: the property is created by passing the *function object f* (and is created at the time when the `class` statement for `B` executes, so the function object in question is the one also known as *B.f*). The fact that the *name f* is later

redefined in subclass *C* is therefore quite irrelevant, since the property performs no lookup for that name, but rather uses the function object it was passed at creation time. If you need to work around this issue, you can always do it with one extra level of indirection:

```
class B(object):
  def f(self): return 23
  def _f_getter(self): return self.f( )
  g = property(_f_getter)
class C(B):
  def f(self): return 42
c = C( )
print c.g                # prints 42, as expected
```

Here, the function object held by the property is *B._f_getter*, which in turn does perform a lookup for name *f* (since it calls `self.f( )`); therefore, the overriding of f has the expected effect.

## __slots__

Normally, each instance object *x* of any class *C* has a dictionary *x*.__dict__ that Python uses to let you bind arbitrary attributes on *x*. To save a little memory (at the cost of letting *x* have only a predefined set of attribute names), you can define in a new-style class *C* a class attribute named __slots__, a sequence (normally a tuple) of strings (normally identifiers). When a new-style class *C* has an attribute __slots__, a direct instance *x* of class *C* has no *x*.__dict__, and any attempt to bind on *x* any attribute whose name is not in *C*.__slots__ raises an exception. Using __slots__ lets you reduce memory consumption for small instance objects that can do without the powerful and convenient ability to have arbitrarily named attributes. __slots__ is worth adding only to classes that can have so many instances that saving a few tens of bytes per instance is important—typically classes that can have millions, not mere thousands, of instances alive at the same time. Unlike most other class attributes, __slots__ works as I've just described only if some statement in the class body binds it as a class attribute. Any later alteration, rebinding, or unbinding of __slots__ has no effect, nor does inheriting __slots__ from a base class. Here's how to add __slots__ to the Rectangle class defined earlier to get smaller (though less flexible) instances:

```
class OptimizedRectangle(Rectangle):
    __slots__ = 'width', 'height'
```

We do not need to define a slot for the area property. __slots__ does not constrain properties, only ordinary instance attributes, which are the attributes that would reside in the instance's __dict__ if __slots__ wasn't defined.

## __getattribute__

All references to instance attributes for new-style instances proceed through special method __getattribute__. This method is supplied by base class object, where it implements all the details of object attribute reference semantics documented in "Attribute Reference Basics" on page 89. However, you may override __getattribute__ for special purposes, such as hiding inherited class attributes

(e.g., methods) for your subclass's instances. The following example shows one way to implement a list without append in the new-style object model:

```
class listNoAppend(list):
    def __getattribute__(self, name):
        if name == 'append': raise AttributeError, name
        return list.__getattribute__(self, name)
```

An instance *x* of class listNoAppend is almost indistinguishable from a built-in list object, except that performance is substantially worse, and any reference to *x*.append raises an exception.

## Per-Instance Methods

Both the legacy and new-style object models allow an instance to have instance-specific bindings for all attributes, including callable attributes (methods). For a method, just like for any other attribute (except those bound to overriding descriptors in new-style classes), an instance-specific binding hides a class-level binding: attribute lookup does not consider the class when it finds a binding directly in the instance. In both object models, an instance-specific binding for a callable attribute does not perform any of the transformations detailed in "Bound and Unbound Methods" on page 91. In other words, the attribute reference returns exactly the same callable object that was earlier bound directly to the instance attribute.

Legacy and new-style object models do differ on the effects of per-instance bindings of the special methods that Python invokes implicitly as a result of various operations, as covered in "Special Methods" on page 104. In the classic object model, an instance may usefully override a special method, and Python uses the per-instance binding even when invoking the method implicitly. In the new-style object model, implicit use of special methods always relies on the class-level binding of the special method, if any. The following code shows this difference between the legacy and new-style object models:

```
def fakeGetItem(idx): return idx
class Classic: pass
c = Classic( )
c.__getitem__ = fakeGetItem
print c[23]                         # prints: 23
class NewStyle(object): pass
n = NewStyle( )
n.__getitem__ = fakeGetItem
print n[23]                         # results in:
# Traceback (most recent call last):
#   File "<stdin>", line 1, in ?
# TypeError: unindexable object
```

The semantics of the classic object model in this regard are sometimes handy for tricky and somewhat obscure purposes. However, the new-style object model's approach is more general, and it regularizes and simplifies the relationship between classes and metaclasses, covered in "Metaclasses" on page 116.

### Inheritance from Built-in Types

A new-style class can inherit from a built-in type. However, a class may directly or indirectly subclass multiple built-in types only if those types are specifically designed to allow this level of mutual compatibility. Python does not support unconstrained inheritance from multiple arbitrary built-in types. Normally, a new-style class only subclasses at most one substantial built-in type—this means at most one built-in type in addition to `object`, which is the superclass of all built-in types and new-style classes and imposes no constraints on multiple inheritance. For example:

```
class noway(dict, list): pass
```

raises a `TypeError` exception, with a detailed explanation of "Error when calling the metaclass bases: multiple bases have instance lay-out conflict." If you ever see such error messages, it means that you're trying to inherit, directly or indirectly, from multiple built-in types that are not specifically designed to cooperate at such a deep level.

## Special Methods

A class may define or inherit special methods (i.e., methods whose names begin and end with double underscores). Each special method relates to a specific operation. Python implicitly invokes a special method whenever you perform the related operation on an instance object. In most cases, the method's return value is the operation's result, and attempting an operation when its related method is not present raises an exception. Throughout this section, I will point out the cases in which these general rules do not apply. In the following, *x* is the instance of class *C* on which you perform the operation, and *y* is the other operand, if any. The formal argument `self` of each method also refers to instance object *x*. Whenever, in the following sections, I mention calls to *x*`.__`*name*`__(...)`, keep in mind that, for new-style classes, the exact call happening is rather, pedantically speaking, *x*`.__class__.__`*name*`__(`*x*`, ...)`.

### General-Purpose Special Methods

Some special methods relate to general-purpose operations. A class that defines or inherits these methods allows its instances to control such operations. These operations can be divided into the following categories:

*Initialization and finalization*
: A class can control its instances' initialization (a frequent need) via special methods `__new__` (new-style classes only) and `__init__`, and/or their finalization (a rare need) via `__del__`.

*Representation as string*
: A class can control how Python represents its instances as strings via special methods `__repr__`, `__str__`, and `__unicode__`.

*Comparison, hashing, and use in a Boolean context*
: A class can control how its instances compare with other objects (methods `__lt__`, `__le__`, `__gt__`, `__ge__`, `__eq__`, `__ne__`, and `__cmp__`), how

dictionaries use them as keys and sets as members (`__hash__`), and whether they evaluate to true or false in Boolean contexts (`__nonzero__`).

*Attribute reference, binding, and unbinding*
: A class can control access to its instances' attributes (reference, binding, unbinding) via special methods `__getattribute__` (new-style classes only), `__getattr__`, `__setattr__`, and `__delattr__`.

*Callable instances*
: An instance is callable, just like a function object, if its class has the special method `__call__`.

The rest of this section documents the general-purpose special methods.

---

**`__call__`** `__call__(self[,args...])`

When you call `x([args...])`, Python translates the operation into a call to `x.__call__([args...])`. The parameters for the call operation are the same as for the `__call__` method, minus the first. The first parameter, conventionally called `self`, refers to `x`, and Python supplies it implicitly and automatically, just as in any other call to a bound method.

---

**`__cmp__`** `__cmp__(self,other)`

Any comparison operator, when its specific special method (`__lt__`, `__gt__`, etc.) is absent or returns `NotImplemented`, calls `x.__cmp__(y)` instead, as do built-in functions requiring comparisons, such as `cmp(x, y)`, `max(x, y)`, and the `sort` method of list objects. `__cmp__` should return `-1` if `x` is less than `y`, `0` if `x` is equal to `y`, or `1` if `x` is greater than `y`. When `__cmp__` is also absent, order comparisons (`<`, `<=`, `>`, `>=`) raise exceptions. Equality comparisons (`==`, `!=`), in this case, become identity checks: `x==y` evaluates `id(x)==id(y)` (i.e., `x is y`).

---

**`__del__`** `__del__(self)`

Just before `x` disappears because of garbage collection, Python calls `x.__del__()` to let `x` finalize itself. If `__del__` is absent, Python performs no special finalization upon garbage-collecting `x` (this is the usual case, as very few classes need to define `__del__`). Python ignores the return value of `__del__`. Python performs no implicit call to `__del__` methods of class `C`'s superclasses. `C.__del__` must explicitly perform any needed finalization.

For example, when class `C` has a base class `B` to finalize, the code in `C.__del__` must call `B.__del__(self)` (or better, for new-style classes, `super(C, self).__del__()`). `__del__` is generally not the best approach when you need timely and guaranteed finalization. For such needs, use the `try/finally` statement covered in "try/finally" (or, even better, in Python 2.5, the new `with` statement, covered in "The with statement" on page 125).

Instances of classes defining `__del__` cannot participate in cyclic-garbage collection, covered in "Garbage Collection" on page 332. Therefore, you should be particularly careful to avoid reference loops involving such instances, and define `__del__` only when there is no reasonable alternative.

**`__delattr__`**

`__delattr__(self, name)`

At every request to unbind attribute `x.y` (typically, a `del` statement `del x.y`), Python calls `x.__delattr__('y')`. All the considerations discussed later for `__setattr__` also apply to `__delattr__`. Python ignores the return value of `__delattr__`. If `__delattr__` is absent, Python usually translates `del x.y` into `del x.__dict__['y']`.

**`__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, `__ne__`**

`__eq__(self, other)` `__ge__(self, other)` `__gt__(self, other)` `__le__(self, other)` `__lt__(self, other)` `__ne__(self, other)`

Comparisons `x==y`, `x>=y`, `x>y`, `x<=y`, `x<y`, and `x!=y`, respectively, call the special methods listed here, which should return `False` or `True`. Each method may return `NotImplemented` to tell Python to handle the comparison in alternative ways (e.g., Python may then try `y>x` in lieu of `x<y`).

**`__getattr__`**

`__getattr__(self, name)`

When attribute `x.y` is accessed but not found by the usual steps (i.e., where `AttributeError` would normally be raised), Python calls `x.__getattr__('y')` instead. Python does not call `__getattr__` for attributes found by normal means (i.e., as keys in `x.__dict__` or via `x.__class__`). If you want Python to call `__getattr__` on every attribute reference, keep the attributes elsewhere (e.g., in another dictionary referenced by an attribute with a private name), or else write a new-style class and override `__getattribute__` instead. `__getattr__` should raise `AttributeError` if it cannot find `y`.

**`__getattribute__`**

`__getattribute__(self, name)`

At every request to access attribute `x.y`, if `x` is an instance of new-style class `C`, Python calls `x.__getattribute__('y')`, which must obtain and return the attribute value or else raise `AttributeError`. The normal semantics of attribute access (using `x.__dict__`, `C.__slots__`, `C`'s class attributes, `x.__getattr__`) are all due to `object.__getattribute__`.

If class `C` overrides `__getattribute__`, it must implement all of the attribute access semantics it wants to offer. Most often, the most convenient way to implement attribute access semantics is by delegating (e.g., calling `object.__getattribute__(self, ...)` as part of the operation of your override of `__getattribute__`). Note that

when a class overrides `__getattribute__`, attribute accesses on instances of the class become slow, since the overriding code is called on every such attribute access.

**`__hash__`** `__hash__(self)`

The `hash(x)` built-in function call, and the use of *x* as a dictionary key (such as `D[x]`, where `D` is a dictionary) or a set member, call `x.__hash__()`. `__hash__` must return a 32-bit `int` such that `x==y` implies `hash(x)==hash(y)`, and must always return the same value for a given object.

When `__hash__` is absent, `hash(x)`, and the use of *x* as a dictionary key or a set member, call `id(x)` instead, as long as `__cmp__` and `__eq__` are also absent.

Any *x* such that `hash(x)` returns a result, rather than raising an exception, is known as a *hashable object*. When `__hash__` is absent, but `__cmp__` or `__eq__` is present, `hash(x)`, and the use of *x* as a dictionary key, raise an exception. In this case, *x* is not hashable and cannot be a dictionary key. Note that `__hash__` is present even when it's not coded directly in the object's class but inherited from a base class. For example, a new-style class which subclasses `object` inherits a `__hash__` which unconditionally calls `id`.

You normally define `__hash__` only for immutable objects that also define `__cmp__` and/or `__eq__`. Note that if there exists any *y* such that `x==y`, even if *y* is of a different type, and both *x* and *y* are hashable, you *must* ensure that `hash(x)==hash(y)`.

**`__init__`** `__init__(self[,args...])`

When a call `C([args...])` creates instance *x* of class `C`, Python calls `x.__init__([args...])` to let *x* initialize itself. If `__init__` is absent, you must call class `C` without arguments, `C()`, and *x* has no instance-specific attributes upon creation. Strictly speaking, `__init__` is never absent for a new-style class `C`, since such a class inherits `__init__` from `object` unless it redefines it; however, even in this case, you must still call class `C` without arguments, `C()`, and the resulting instance has no instance-specific attributes upon creation.

`__init__` must return `None`. Python performs no implicit call to `__init__` methods of class `C`'s superclasses. `C.__init__` must explicitly perform any needed initialization. For example, when class `C` has a base class `B` to initialize without arguments, the code in `C.__init__` must explicitly call `B.__init__(self)` (or better, for new-style classes, `super(C, self).__init__()`).

**`__new__`** `__new__(cls[,args...])`

When you call `C([args...])` and `C` is a new-style class, Python obtains the new instance *x* that you are creating by invoking `C.__new__(C,[args...])`. `__new__` is a static method that every new-

style class has (often simply inheriting it from `object`) and it can return any value *x*. In other words, `__new__` is not constrained to return a new instance of *C*, although normally it's expected to do so. If, and only if, the value *x* that `__new__` returns is indeed an instance of *C* (whether a new or previously existing one), Python continues after calling `__new__` by implicitly calling `__init__` on *x* (with the same [*args*...] that were originally passed to `__new__`).

Since you could perform most kinds of initialization on new instances in either special method, `__init__` or `__new__`, you may wonder where it's best to place them. The answer is simple: put every kind of initialization in `__init__` only, unless you have some specific, advanced reason to put some in `__new__` instead. This will make your life much simpler in all kinds of situations, due to the fact that `__init__` is an instance method while `__new__` is a rather specialized static method.

---

**__nonzero__** `__nonzero__(self)`

When evaluating `x` as true or false (see "Boolean Values" on page 45)—for example, on a call to `bool(x)`—Python calls *x*`.__nonzero__()`, which should return `True` or `False`. When `__nonzero__` is not present, Python calls `__len__` instead, and takes *x* as false when *x*`.__len__()` returns `0` (so, to check if a container is nonempty, avoid coding `if len(`*container*`)>0:`; just code `if` *container*`:` instead). When neither `__nonzero__` nor `__len__` is present, Python always considers *x* true.

---

**__repr__** `__repr__(self)`

The `repr(`*x*`)` built-in function call, the `` `x` `` expression form, and the interactive interpreter (when *x* is the result of an expression statement) call *x*`.__repr__()` to obtain an "official," complete string representation of *x*. If `__repr__` is absent, Python uses a default string representation. `__repr__` should return a string with unambiguous information on *x*. Ideally, when feasible, the string should be an expression such that `eval(repr(`*x*`))==`*x*.

---

**__setattr__** `__setattr__(self, name, value)`

At every request to bind attribute *x*`.`*y* (typically, an assignment statement *x*`.`*y*`=`*value*), Python calls *x*`.__setattr__('`*y*`',` *value*`)`. Python always calls `__setattr__` for *any* attribute binding on *x*—a major difference from `__getattr__` (`__setattr__` is closer to new-style classes' `__getattribute__` in this sense). To avoid recursion, when *x*`.__setattr__` binds *x*'s attributes, it must modify *x*`.__dict__` directly (e.g., via *x*`.__dict__[`*name*`]=`*value*); even better, for a new-style class, `__setattr__` can delegate the setting to the superclass (by calling `super(C, x).__setattr__('`*y*`',` *value*`)`). Python ignores the return value of `__setattr__`. If `__setattr__` is absent, Python usually translates *x*`.`*y*`=`*z* into *x*`.__dict__['`*y*`']=`*z*.

**__str__** `__str__(self)`

The `str(x)` built-in type and the `print x` statement call `x.__str__()` to obtain an informal, concise string representation of `x`. If `__str__` is absent, Python calls `x.__repr__` instead. `__str__` should return a conveniently human-readable string, even if it entails some approximation.

**__unicode__** `__unicode__(self)`

The `unicode(x)` built-in type call invokes `x.__unicode__()`, if present, in preference to `x.__str__()`. If a class supplies both special methods `__unicode__` and `__str__`, the two should return equivalent strings (of Unicode and plain-string type, respectively).

## Special Methods for Containers

An instance can be a *container* (either a sequence or a mapping, but not both, as they are mutually exclusive concepts). For maximum usefulness, containers should provide not just special methods `__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `__contains__`, and `__iter__`, but also a few nonspecial methods, as discussed in the following sections.



### Sequences

In each item-access special method, a sequence that has `L` items should accept any integer *key* such that `-L<=key<L`. For compatibility with built-in sequences, a negative index *key*, `0>key>=-L` should be equivalent to *key*`+L`. When *key* has an invalid type, the method should raise `TypeError`. When *key* is a value of a valid type, but out of range, the method should raise `IndexError`. For container classes that do not define `__iter__`, the `for` statement relies on these requirements, as do built-in functions that take iterable arguments. Every item-access special method of sequence should also accept as its index argument an instance of the built-in type `slice` whose `start`, `step`, and `stop` attributes are `int`s or `None`. The *slicing* syntax relies on this requirement, as covered in "Container slicing" on page 110.

A sequence should also allow concatenation (with another sequence of the same type) by `+` and repetition by `*` (multiplication by an integer). A sequence should therefore have special methods `__add__`, `__mul__`, `__radd__`, and `__rmul__`, covered in "Special Methods for Numeric Objects" on page 113. A sequence should be meaningfully comparable to another sequence of the same type, implementing *lexicographic* comparison like lists and tuples do. Mutable sequences should also have `__iadd__` and `__imul__`, and the nonspecial methods covered in "List methods" on page 56: `append`, `count`, `index`, `insert`, `extend`, `pop`, `remove`, `reverse`, and `sort`, with the same signatures and semantics as the corresponding methods of lists. An immutable sequence should be hashable if all of its items are.

A sequence type may constrain its items in some ways (for example, by accepting only string items), but that is not mandatory.

#### Mappings

A mapping's item-access special methods should raise `KeyError`, rather than `IndexError`, when they receive an invalid *key* argument value of a valid type. Any mapping should define the nonspecial methods covered in "Dictionary Methods" on page 60: `copy`, `get`, `has_key`, `items`, `keys`, `values`, `iteritems`, `iterkeys`, and `itervalues`. Special method `__iter__` should be equivalent to `iterkeys`. A mapping should be meaningfully comparable to another mapping of the same type. A mutable mapping should also define methods `clear`, `popitem`, `setdefault`, and `update`, while an immutable mapping should be hashable if all of its items are. A mapping type may constrain its keys in some ways (for example, by accepting only hashable keys, or, even more specifically, accepting, say, only string keys), but that is not mandatory.

#### Sets

Sets can be seen as rather peculiar kinds of containers—containers that are neither sequences nor mappings and cannot be indexed, but do have a length (number of elements) and are iterable. Sets also support many operators (`&`, `|`, `^`, `-`, as well as membership tests and comparisons) and equivalent nonspecial methods (`intersection`, `union`, and so on). If you implement a set-like container, it should be polymorphic to Python built-in sets, covered in "Sets" on page 43. An immutable set-like type should be hashable if all of its elements are. A set-like type may constrain its elements in some ways (for example, by accepting only hashable elements, or, even more specifically, accepting, say, only integer elements), but that is not mandatory.

#### Container slicing

When you reference, bind, or unbind a slicing such as *x*`[`*i*`:`*j*`]` or *x*`[`*i*`:`*j*`:`*k*`]` on a container *x*, Python calls *x*'s applicable item-access special method, passing as *key* an object of a built-in type called a *slice object*. A slice object has attributes `start`, `stop`, and `step`. Each attribute is `None` if the corresponding value is omitted in the slice syntax. For example, `del` *x*`[:3]` calls *x*`.__delitem__(`*y*`)`, and *y* is a slice object such that *y*`.stop` is `3`, *y*`.start` is `None`, and *y*`.step` is `None`. It is up to container object *x* to appropriately interpret the slice object argument passed to *x*'s special methods. Method `indices` of slice objects can help: call it with your container's length as its only argument, and it returns a tuple of three nonnegative indices suitable as start, stop, and step for a loop indexing each item in the slice. A common idiom in a sequence class's `__getitem__` special method, to fully support slicing, might be:

```
def __getitem__(self, index):
  # recursively specialcase slicing
  if isinstance(index, slice):
    return self.__class__(self[x]
                          for x in xrange(*index.indices(len(self))))
  # check index, dealing with negative indices too
```

```
if not isinstance(index, int): raise TypeError
if index<0: index+=len(self)
if not (0<=index<len(self)): raise IndexError
# index is now a correct int, within range(len(self))
...rest of __getitem__, dealing with single-item access by int index...
```

This idiom uses Python 2.4 generator-expression (genexp) syntax and assumes that your class's `__init__` method can be called with an iterable argument to create a suitable new instance of the class.

Some built-in types, such as `list` and `tuple`, define (for reasons of backward compatibility) now-deprecated special methods `__getslice__`, `__setslice__`, and `__delslice__`. For an instance *x* of such a type, slicing *x* with only one colon, as in *x*`[`*i*`:`*j*`]`, calls a slice-specific special method. Slicing *x* with two colons, as in *x*`[`*i*`:`*j*`:`*k*`]`, calls an item-access special method with a slice object argument. For example:

```
class C:
    def __getslice__(self, i, j): print 'getslice', i, j
    def __getitem__(self, index): print 'getitem', index
x = C()
x[12:34]
x[56:78:9]
```

The first slicing calls *x*`.__getslice__(12,34)`, while the second calls *x*`.__getitem__(slice(56,78,9))`. It's best to avoid this complication by simply *not* defining the slice-specific special methods in your classes; however, you may need to override these methods if your class subclasses `list` or `tuple` and you want to provide special functionality when an instance of your class is sliced with just one colon.



### Container methods

Special methods `__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`, and `__contains__` expose container functionality.

**`__contains__`** `__contains__(self,`*item*`)`

The Boolean test *y* `in` *x* calls *x*`.__contains__(`*y*`)`. When *x* is a sequence, `__contains__` should return `True` when *y* equals the value of an item in the sequence. When *x* is a mapping, `__contains__` should return `True` when *y* equals the value of a key in the mapping. Otherwise, `__contains__` should return `False`. If `__contains__` is absent, Python performs *y* `in` *x* as follows, taking time proportional to `len(`*x*`)`:

```
for z in x:
    if y==z: return True
return False
```

**__delitem__** `__delitem__(self,key)`

For a request to unbind an item or slice of *x* (typically `del x[key]`), Python calls `x.__delitem__(key)`. A container *x* should have `__delitem__` only if *x* is mutable so that items (and possibly slices) can be removed.

**__getitem__** `__getitem__(self,key)`

When `x[key]` is accessed (i.e., when container *x* is indexed or sliced), Python calls `x.__getitem__(key)`. All (non-set-like) containers should have `__getitem__`.

**__iter__** `__iter__(self)`

For a request to loop on all items of *x* (typically `for item in x`), Python calls `x.__iter__()` to obtain an iterator on *x*. The built-in function `iter(x)` also calls `x.__iter__()`. When `__iter__` is absent and *x* is a sequence, `iter(x)` synthesizes and returns an iterator object that wraps *x* and returns `x[0]`, `x[1]`, and so on until one of these indexings raises `IndexError` to indicate the end of the sequence. However, it is best to ensure that all of the container classes you code have `__iter__`.

**__len__** `__len__(self)`

The `len(x)` built-in function call, and other built-in functions that need to know how many items are in container *x*, call `x.__len__()`. `__len__` should return an `int`, the number of items in *x*. Python also calls `x.__len__()` to evaluate *x* in a Boolean context, when `__nonzero__` is absent. Absent `__nonzero__`, a container is taken as false if and only if the container is empty (i.e., the container's length is `0`). All containers should have `__len__`, unless it's exceedingly expensive for the container to determine how many items it currently contains.

**__setitem__** `__setitem__(self,key,value)`

For a request to bind an item or slice of *x* (typically an assignment `x[key]=value`), Python calls `x.__setitem__(key,value)`. A container *x* should have `__setitem__` only if *x* is mutable so that items, and possibly slices, can be added and/or rebound.

## Special Methods for Numeric Objects

An instance may support numeric operations by means of many special methods. Some classes that are not numbers also support some of the following special methods in order to overload operators such as + and *. For example, sequences should have special methods `__add__`, `__mul__`, `__radd__`, and `__rmul__`, as mentioned in "Sequences" on page 109.

---

**`__abs__`, `__invert__`, `__neg__`, `__pos__`**

`__abs__(self)` `__invert__(self)` `__neg__(self)` `__pos__(self)`

Unary operators `abs(x)`, `~x`, `-x`, and `+x`, respectively, call these methods.

---

**`__add__`, `__div__`, `__floordiv__`, `__mod__`, `__mul__`, `__sub__`, `__truediv__`**

`__add__(self,other)` `__div__(self,other)` `__floordiv__(self,other)` `__mod__(self,other)` `__mul__(self,other)` `__sub__(self,other)` `__truediv__(self,other)`

Operators *x*+*y*, *x*/*y*, *x*//*y*, *x*%*y*, *x**y*, *x*-*y*, and *x*/*y*, respectively, call these methods. The operator / calls `__truediv__`, if present, instead of `__div__`, in the situations where division is nontruncating, as covered in "Arithmetic Operations" on page 52.

---

**`__and__`, `__lshift__`, `__or__`, `__rshift__`, `__xor__`**

`__and__(self,other)` `__lshift__(self,other)` `__or__(self,other)` `__rshift__(self,other)` `__xor__(self,other)`

Operators *x*&*y*, *x*<<*y*, *x*|*y*, *x*>>*y*, and *x*^*y*, respectively, call these methods.

---

**`__coerce__`**

`__coerce__(self,other)`

For any numeric operation with two operands *x* and *y*, Python invokes *x*.`__coerce__`(*y*). `__coerce__` should return a pair with *x* and *y* converted to acceptable types. `__coerce__` returns `None` when it cannot perform the conversion. In such cases, Python calls *y*.`__coerce__`(*x*). This special method is now deprecated; your classes should not implement it, but instead deal with whatever types they can accept directly in the special methods of the relevant numeric operations. However, if a class does supply `__coerce__`, Python still calls it for backward compatibility.

---

**`__complex__`, `__float__`, `__int__`, `__long__`**

`__complex__(self)` `__float__(self)` `__int__(self)` `__long__(self)`

Built-in types `complex(x)`, `float(x)`, `int(x)`, and `long(x)`, respectively, call these methods.