3

Types and Objects

ALL THE DATA STORED IN A PYTHON program is built around the concept of an *object*. Objects include fundamental data types such as numbers, strings, lists, and dictionaries. It's also possible to create user-defined objects in the form of classes or extension types. This chapter describes the Python object model and provides an overview of the built-in data types. Chapter 4, "Operators and Expressions," further describes operators and expressions.

Terminology

Every piece of data stored in a program is an object. Each object has an identity, a type, and a value.

For example, when you write `a = 42`, an integer object is created with the value of `42`. You can view the *identity* of an object as a pointer to its location in memory. `a` is a name that refers to this specific location.

The *type* of an object (which is itself a special kind of object) describes the internal representation of the object as well as the methods and operations that it supports. When an object of a particular type is created, that object is sometimes called an *instance* of that type. After an object is created, its identity and type cannot be changed. If an object's value can be modified, the object is said to be *mutable*. If the value cannot be modified, the object is said to be *immutable*. An object that contains references to other objects is said to be a *container* or *collection*.

In addition to holding a value, many objects define a number of data attributes and methods. An *attribute* is a property or value associated with an object. A *method* is a function that performs some sort of operation on an object when the method is invoked. Attributes and methods are accessed using the dot (`.`) operator, as shown in the following example:

```
a = 3 + 4j           # Create a complex number
r = a.real           # Get the real part (an attribute)

b = [1, 2, 3]        # Create a list
b.append(7)          # Add a new element using the append method
```

Object Identity and Type

The built-in function `id()` returns the identity of an object as an integer. This integer usually corresponds to the object's location in memory, although this is specific to the

Python implementation and the platform being used. The `is` operator compares the identity of two objects. The built-in function `type()` returns the type of an object. For example:

```
# Compare two objects
def compare(a,b):
    print 'The identity of a is ', id(a)
    print 'The identity of b is ', id(b)
    if a is b:
        print 'a and b are the same object'
    if a == b:
        print 'a and b have the same value'
    if type(a) is type(b):
        print 'a and b have the same type'
```

The type of an object is itself an object. This type object is uniquely defined and is always the same for all instances of a given type. Therefore, the type can be compared using the `is` operator. All type objects are assigned names that can be used to perform type checking. Most of these names are built-ins, such as `list`, `dict`, and `file`. For example:

```
if type(s) is list:
    print 'Is a list'

if type(f) is file:
    print 'Is a file'
```

However, some type names are only available in the `types` module. For example:

```
import types
if type(s) is types.NoneType:
    print "is None"
```

Because types can be specialized by defining classes, a better way to check types is to use the built-in `isinstance(`*`object, type`*`)` function. For example:

```
if isinstance(s,list):
    print 'Is a list'

if isinstance(f,file):
    print 'Is a file'

if isinstance(n,types.NoneType):
    print "is None"
```

The `isinstance()` function also works with user-defined classes. Therefore, it is a generic, and preferred, way to check the type of any Python object.

Reference Counting and Garbage Collection

All objects are reference-counted. An object's reference count is increased whenever it's assigned to a new name or placed in a container such as a list, tuple, or dictionary, as shown here:

```
a = 3.4       # Creates an object '3.4'
b = a         # Increases reference count on '3.4'
c = []
c.append(b)   # Increases reference count on '3.4'
```

This example creates a single object containing the value `3.4`. `a` is merely a name that refers to the newly created object. When `b` is assigned `a`, `b` becomes a new name for the same object, and the object's reference count increases. Likewise, when you place `b` into a list, the object's reference count increases again. Throughout the example, only one object contains `3.4`. All other operations are simply creating new references to the object.

An object's reference count is decreased by the `del` statement or whenever a reference goes out of scope (or is reassigned). For example:

```
del a       # Decrease reference count of 3.4
b = 7.8     # Decrease reference count of 3.4
c[0]=2.0    # Decrease reference count of 3.4
```

When an object's reference count reaches zero, it is garbage-collected. However, in some cases a circular dependency may exist among a collection of objects that are no longer in use. For example:

```
a = { }
b = { }
a['b'] = b      # a contains reference to b
b['a'] = a      # b contains reference to a
del a
del b
```

In this example, the `del` statements decrease the reference count of `a` and `b` and destroy the names used to refer to the underlying objects. However, because each object contains a reference to the other, the reference count doesn't drop to zero and the objects remain allocated (resulting in a memory leak). To address this problem, the interpreter periodically executes a cycle-detector that searches for cycles of inaccessible objects and deletes them. The cycle-detection algorithm can be fine-tuned and controlled using functions in the `gc` module.

References and Copies

When a program makes an assignment such as `a = b`, a new reference to `b` is created. For immutable objects such as numbers and strings, this assignment effectively creates a copy of `b`. However, the behavior is quite different for mutable objects such as lists and dictionaries. For example:

```
b = [1,2,3,4]
a = b                # a is a reference to b
a[2] = -100          # Change an element in 'a'
print b              # Produces '[1, 2, -100, 4]'
```

Because `a` and `b` refer to the same object in this example, a change made to one of the variables is reflected in the other. To avoid this, you have to create a copy of an object rather than a new reference.

Two types of copy operations are applied to container objects such as lists and dictionaries: a shallow copy and a deep copy. A *shallow copy* creates a new object, but populates it with references to the items contained in the original object. For example:

```
b = [ 1, 2, [3,4] ]
a = b[:]              # Create a shallow copy of b.
a.append(100)         # Append element to a.
print b               # Produces '[1,2, [3,4]]'. b unchanged.
a[2][0] = -100        # Modify an element of a.
print b               # Produces '[1,2, [-100,4]]'.
```

In this case, a and b are separate list objects, but the elements they contain are shared. Therefore, a modification to one of the elements of a also modifies an element of b, as shown.

A *deep copy* creates a new object and recursively copies all the objects it contains. There is no built-in function to create deep copies of objects. However, the copy.deepcopy() function in the standard library can be used, as shown in the following example:

```
import copy
b = [1, 2, [3, 4] ]
a = copy.deepcopy(b)

a[2] = -100

print a    # produces [1,2, -100, 4]

print b    # produces [1,2,3,4]
```

Built-in Types

Approximately two dozen types are built into the Python interpreter and grouped into a few major categories, as shown in Table 3.1. The Type Name column in the table lists the name that can be used to check for that type using isinstance() and other type-related functions. Types include familiar objects such as numbers and sequences. Others are used during program execution and are of little practical use to most programmers. The next few sections describe the most commonly used built-in types.

Table 3.1 **Built-in Python Types**

Type Category	Type Name	Description
None	types.NoneType	The null object None
Numbers	int	Integer
	long	Arbitrary-precision integer
	float	Floating point
	complex	Complex number
	bool	Boolean (True or False)
Sequences	str	Character string
	unicode	Unicode character string
	basestring	Abstract base type for all strings
	list	List
	tuple	Tuple
	xrange	Returned by xrange()
Mapping	dict	Dictionary
Sets	set	Mutable set
	frozenset	Immutable set

Table 3.1 **Continued**

Type Category	Type Name	Description
Callable	`types.BuiltinFunctionType`	Built-in functions
	`types.BuiltinMethodType`	Built-in methods
	`type`	Type of built-in types and classes
	`object`	Ancestor of all types and classes
	`types.FunctionType`	User-defined function
	`types.InstanceType`	Class object instance
	`types.MethodType`	Bound class method
	`types.UnboundMethodType`	Unbound class method
Modules	`types.ModuleType`	Module
Classes	`object`	Ancestor of all types and classes
Types	`type`	Type of built-in types and classes
Files	`file`	File
Internal	`types.CodeType`	Byte-compiled code
	`types.FrameType`	Execution frame
	`types.GeneratorType`	Generator object
	`types.TracebackType`	Stacks traceback of an exception
	`types.SliceType`	Generated by extended slices
	`types.EllipsisType`	Used in extended slices
Classic Classes	`types.ClassType`	Old-style class definition
	`types.InstanceType`	Old-style class instance

Note that `object` and `type` appear twice in Table 3.1 because classes and types are both callable. The types listed for "Classic Classes" refer to an obsolete, but still supported object-oriented interface. More details about this can be found later in this chapter and in Chapter 7, "Classes and Object-Oriented Programming."

The `None` Type

The `None` type denotes a null object (an object with no value). Python provides exactly one null object, which is written as `None` in a program. This object is returned by functions that don't explicitly return a value. None is frequently used as the default value of optional arguments, so that the function can detect whether the caller has actually passed a value for that argument. `None` has no attributes and evaluates to `False` in Boolean expressions.

Numeric Types

Python uses five numeric types: Booleans, integers, long integers, floating-point numbers, and complex numbers. Except for Booleans, all numeric objects are signed. All numeric types are immutable.

Booleans are represented by two values: `True` and `False`. The names `True` and `False` are respectively mapped to the numerical values of 1 and 0.

Integers represent whole numbers in the range of –2147483648 to 2147483647 (the range may be larger on some machines). Internally, integers are stored as 2's complement binary values, in 32 or more bits. Long integers represent whole numbers of unlimited range (limited only by available memory). Although there are two integer types, Python tries to make the distinction seamless. Most functions and operators that expect integers work with any integer type. Moreover, if the result of a numerical operation exceeds the allowed range of integer values, the result is transparently promoted to a long integer (although in certain cases, an `OverflowError` exception may be raised instead).

Floating-point numbers are represented using the native double-precision (64-bit) representation of floating-point numbers on the machine. Normally this is IEEE 754, which provides approximately 17 digits of precision and an exponent in the range of –308 to 308. This is the same as the `double` type in C. Python doesn't support 32-bit single-precision floating-point numbers. If space and precision are an issue in your program, consider using Numerical Python (http://numpy.sourceforge.net).

Complex numbers are represented as a pair of floating-point numbers. The real and imaginary parts of a complex number *z* are available in *z*`.real` and *z*`.imag`.

Sequence Types

Sequences represent ordered sets of objects indexed by nonnegative integers and include strings, Unicode strings, lists, and tuples. Strings are sequences of characters, and lists and tuples are sequences of arbitrary Python objects. Strings and tuples are immutable; lists allow insertion, deletion, and substitution of elements. All sequences support iteration.

Table 3.2 shows the operators and methods that you can apply to all sequence types. Element *i* of sequence *s* is selected using the indexing operator *s*`[`*i*`]`, and subsequences are selected using the slicing operator *s*`[`*i*`:`*j*`]` or extended slicing operator *s*`[`*i*`:`*j*`:`*stride*`]` (these operations are described in Chapter 4). The length of any sequence is returned using the built-in `len(`*s*`)` function. You can find the minimum and maximum values of a sequence by using the built-in `min(`*s*`)` and `max(`*s*`)` functions. However, these functions only work for sequences in which the elements can be ordered (typically numbers and strings).

Table 3.3 shows the additional operators that can be applied to mutable sequences such as lists.

Table 3.2 **Operations and Methods Applicable to All Sequences**

Item	Description
`s[i]`	Returns element *i* of a sequence
`s[i:j]`	Returns a slice
`s[i:j:stride]`	Returns an extended slice
`len(s)`	Number of elements in *s*

Table 3.2 **Continued**

Item	Description
`min(s)`	Minimum value in *s*
`max(s)`	Maximum value in *s*

Table 3.3 **Operations Applicable to Mutable Sequences**

Item	Description
`s[i] = v`	Item assignment
`s[i:j] = t`	Slice assignment
`s[i:j:stride] = t`	Extended slice assignment
`del s[i]`	Item deletion
`del s[i:j]`	Slice deletion
`del s[i:j:stride]`	Extended slice deletion

Additionally, lists support the methods shown in Table 3.4. The built-in function `list(s)` converts any iterable type to a list. If *s* is already a list, this function constructs a new list that's a shallow copy of *s*. The `s.append(x)` method appends a new element, *x*, to the end of the list. The `s.index(x)` method searches the list for the first occurrence of *x*. If no such element is found, a `ValueError` exception is raised. Similarly, the `s.remove(x)` method removes the first occurrence of *x* from the list. The `s.extend(t)` method extends the list *s* by appending the elements in sequence *t*. The `s.sort()` method sorts the elements of a list and optionally accepts a comparison function, key function, and reverse flag. The comparison function should take two arguments and return negative, zero, or positive, depending on whether the first argument is smaller, equal to, or larger than the second argument, respectively. The key function is a function that is applied to each element prior to comparison during sorting. Specifying a key function is useful if you want to perform special kinds of sorting operations, such as sorting a list of strings, but with case insensitivity. The `s.reverse()` method reverses the order of the items in the list. Both the `sort()` and `reverse()` methods operate on the list elements in place and return `None`.

Table 3.4 **List Methods**

Method	Description
`list(s)`	Converts *s* to a list.
`s.append(x)`	Appends a new element, *x*, to the end of *s*.
`s.extend(t)`	Appends a new list, *t*, to the end of *s*.
`s.count(x)`	Counts occurrences of *x* in *s*.
`s.index(x [,start [,stop]])`	Returns the smallest *i* where `s[i] ==x`. *start* and *stop* optionally specify the starting and ending index for the search.
`s.insert(i,x)`	Inserts *x* at index *i*.

Table 3.4 **Continued**

Method	Description
`s.pop([i])`	Returns the element *i* and removes it from the list. If *i* is omitted, the last element is returned.
`s.remove(x)`	Searches for *x* and removes it from *s*.
`s.reverse()`	Reverses items of *s* in place.
`s.sort([cmpfunc [, keyf [, reverse]]])`	Sorts items of *s* in place. *cmpfunc* is a comparison function. *keyf* is a key function. *reverse* is a flag that sorts the list in reverse order.

Python provides two string object types. Standard strings are sequences of bytes containing 8-bit data. They may contain binary data and embedded NULL bytes. Unicode strings are sequences of 16-bit characters encoded in a format known as UCS-2. This allows for 65,536 unique character values. Although the latest Unicode standard supports up to 1 million unique character values, these extra characters are not supported by Python by default. Instead, they must be encoded as a special two-character (4-byte) sequence known as a *surrogate pair*—the interpretation of which is up to the application. Python does not check data for Unicode compliance or the proper use of surrogates. As an optional feature, Python may be built to store Unicode strings using 32-bit integers (UCS-4). When enabled, this allows Python to represent the entire range of Unicode values from U+000000 to U+110000. All Unicode-related functions are adjusted accordingly.

Both standard and Unicode strings support the methods shown in Table 3.5. Although these methods operate on string instances, none of these methods actually modifies the underlying string data. Thus, methods such as `s.capitalize()`, `s.center()`, and `s.expandtabs()` always return a new string as opposed to modifying the string *s*. Character tests such as `s.isalnum()` and `s.isupper()` return `True` or `False` if all the characters in the string *s* satisfy the test. Furthermore, these tests always return `False` if the length of the string is zero. The `s.find()`, `s.index()`, `s.rfind()`, and `s.rindex()` methods are used to search *s* for a substring. All these functions return an integer index to the substring in *s*. In addition, the `find()` method returns `-1` if the substring isn't found, whereas the `index()` method raises a `ValueError` exception. Many of the string methods accept optional *start* and *end* parameters, which are integer values specifying the starting and ending indices in *s*. In most cases, these values may given negative values, in which case the index is taken from the end of the string. The `s.translate()` method is used to perform character substitutions. The `s.encode()` and `s.decode()` methods are used to transform the string data to and from a specified character encoding. As input it accepts an encoding name such as `'ascii'`, `'utf-8'`, or `'utf-16'`. This method is most commonly used to convert Unicode strings into a data encoding suitable for I/O operations and is described further in Chapter 9, "Input and Output." More details about string methods can be found in the documentation for the `string` module.

Table 3.5 **String Methods**

Method	Description
`s.capitalize()`	Capitalizes the first character.
`s.center(width [, pad])`	Centers the string in a field of length *width*. *pad* is a padding character.
`s.count(sub [,start [,end]])`	Counts occurrences of the specified substring *sub*.
`s.decode([encoding [,errors]])`	Decodes a string and returns a Unicode string.
`s.encode([encoding [,errors]])`	Returns an encoded version of the string.
`s.endswith(suffix [,start [,end]])`	Checks the end of the string for a suffix.
`s.expandtabs([tabsize])`	Replaces tabs with spaces.
`s.find(sub [, start [,end]])`	Finds the first occurrence of the specified substring *sub*.
`s.index(sub [, start [,end]])`	Finds the first occurrence or error in the specified substring *sub*.
`s.isalnum()`	Checks whether all characters are alphanumeric.
`s.isalpha()`	Checks whether all characters are alphabetic.
`s.isdigit()`	Checks whether all characters are digits.
`s.islower()`	Checks whether all characters are lowercase.
`s.isspace()`	Checks whether all characters are whitespace.
`s.istitle()`	Checks whether the string is a title-cased string (first letter of each word capitalized).
`s.isupper()`	Checks whether all characters are uppercase.
`s.join(t)`	Joins the strings *s* and *t*.
`s.ljust(width [, fill])`	Left-aligns *s* in a string of size *width*.
`s.lower()`	Converts to lowercase.
`s.lstrip([chrs])`	Removes leading whitespace or characters supplied in *chrs*.
`s.replace(old, new [,maxreplace])`	Replaces the substring.
`s.rfind(sub [,start [,end]])`	Finds the last occurrence of a substring.
`s.rindex(sub [,start [,end]])`	Finds the last occurrence or raises an error.
`s.rjust(width [, fill])`	Right-aligns *s* in a string of length *width*.
`s.rsplit([sep [,maxsplit]])`	Splits a string from the end of the string using *sep* as a delimiter. *maxsplit* is the maximum number of splits to perform. If *maxsplit* is omitted, the result is identical to the `split()` method.
`s.rstrip([chrs])`	Removes trailing whitespace or characters supplied in *chrs*.

Table 3.5 **Continued**

Method	Description
`s.split([sep [,maxsplit]])`	Splits a string using `sep` as a delimiter. `maxsplit` is the maximum number of splits to perform.
`s.splitlines([keepends])`	Splits a string into a list of lines. If `keepends` is 1, trailing newlines are preserved.
`s.startswith(prefix [,start [,end]])`	Checks whether a string starts with `prefix`.
`s.strip([chrs])`	Removes leading and trailing whitespace or characters supplied in `chrs`.
`s.swapcase()`	Converts uppercase to lowercase, and vice versa.
`s.title()`	Returns a title-cased version of the string.
`s.translate(table [,deletechars])`	Translates a string using a character translation table `table`, removing characters in `deletechars`.
`s.upper()`	Converts a string to uppercase.
`s.zill(width)`	Pads a string with zeros on the left up to the specified `width`.

Because there are two different string types, Python provides an abstract type, `basestring`, that can be used to test if an object is any kind of string. Here's an example:

```
if isinstance(s,basestring):
    print "is some kind of string"
```

The built-in function `range([i,]j [,stride])` constructs a list and populates it with integers *k* such that `i <= k < j`. The first index, *i*, and the *stride* are optional and have default values of `0` and `1`, respectively. The built-in `xrange([i,] j [,stride])` function performs a similar operation, but returns an immutable sequence of type `xrange`. Rather than storing all the values in a list, this sequence calculates its values whenever it's accessed. Consequently, it's much more memory-efficient when working with large sequences of integers. However, the `xrange` type is much more limited than its list counterpart. For example, none of the standard slicing operations are supported. This limits the utility of `xrange` to only a few applications such as iterating in simple loops. The `xrange` type provides a single method, `s.tolist()`, that converts its values to a list.

Mapping Types

A *mapping object* represents an arbitrary collection of objects that are indexed by another collection of nearly arbitrary key values. Unlike a sequence, a mapping object is unordered and can be indexed by numbers, strings, and other objects. Mappings are mutable.

Dictionaries are the only built-in mapping type and are Python's version of a hash table or associative array. You can use any immutable object as a dictionary key value (strings, numbers, tuples, and so on). Lists, dictionaries, and tuples containing mutable objects cannot be used as keys (the dictionary type requires key values to remain constant).

To select an item in a mapping object, use the key index operator `m[k]`, where *k* is a key value. If the key is not found, a `KeyError` exception is raised. The `len(m)` function returns the number of items contained in a mapping object. Table 3.6 lists the methods and operations.

Table 3.6 **Methods and Operations for Dictionaries**

Item	Description
`len(m)`	Returns the number of items in *m*.
`m[k]`	Returns the item of *m* with key *k*.
`m[k]=x`	Sets `m[k]` to *x*.
`del m[k]`	Removes `m[k]` from *m*.
`m.clear()`	Removes all items from *m*.
`m.copy()`	Makes a shallow copy of *m*.
`m.has_key(k)`	Returns `True` if *m* has key *k*; otherwise, returns `False`.
`m.items()`	Returns a list of `(key, value)` pairs.
`m.iteritems()`	Returns an iterator that produces `(key, value)` pairs.
`m.iterkeys()`	Returns an iterator that produces dictionary keys.
`m.itervalues()`	Returns an iterator that produces dictionary values.
`m.keys()`	Returns a list of key values.
`m.update(b)`	Adds all objects from *b* to *m*.
`m.values()`	Returns a list of all values in *m*.
`m.get(k [,v])`	Returns `m[k]` if found; otherwise, returns *v*.
`m.setdefault(k [, v])`	Returns `m[k]` if found; otherwise, returns *v* and sets `m[k] = v`.
`m.pop(k [,default])`	Returns `m[k]` if found and removes it from *m*; otherwise, returns *`default`* if supplied or raises `KeyError` if not.
`m.popitem()`	Removes a random `(key, value)` pair from *m* and returns it as a tuple.

The `m.clear()` method removes all items. The `m.copy()` method makes a shallow copy of the items contained in a mapping object and places them in a new mapping object. The `m.items()` method returns a list containing `(key, value)` pairs. The `m.keys()` method returns a list with all the key values, and the `m.values()` method returns a list with all the objects. The `m.update(b)` method updates the current mapping object by inserting all the `(key, value)` pairs found in the mapping object *b*. The `m.get(k [,v])` method retrieves an object, but allows for an optional default value, *v*, that's returned if no such object exists. The `m.setdefault(k [,v])` method is similar to `m.get()`, except that in addition to returning *v* if no object exists, it sets `m[k] = v`. If *v* is omitted, it defaults to `None`. The `m.pop()` method returns an item from a dictionary and removes it at the same time. The `m.popitem()` method is used to iteratively destroy the contents of a dictionary. The `m.iteritems()`, `m.iterkeys()`, and `m.itervalues()` methods return iterators that allow looping over all the dictionary items, keys, or values, respectively.

Set Types

A *set* is an unordered collection of unique items. Unlike sequences, sets provide no indexing or slicing operations. They are also unlike dictionaries in that there are no key values associated with the objects. In addition, the items placed into a set must be immutable. Two different set types are available: `set` is a mutable set, and `frozenset` is an immutable set. Both kinds of sets are created using a pair of built-in functions:

```
s = set([1,5,10,15])
f = frozenset(['a',37,'hello'])
```

Both `set()` and `frozenset()` populate the set by iterating over the supplied argument. Both kinds of sets provide the methods outlined in Table 3.7

Table 3.7 **Methods and Operations for Set Types**

Item	Description
`len(s)`	Return number of items in *s*.
`s.copy()`	Makes a shallow copy of *s*.
`s.difference(t)`	Set difference. Returns all the items in *s*, but not in *t*.
`s.intersection(t)`	Intersection. Returns all the items that are both in *s* and in *t*.
`s.issubbset(t)`	Returns `True` if *s* is a subset of *t*.
`s.issuperset(t)`	Returns `True` if *s* is a superset of *t*.
`s.symmetric_difference(t)`	Symmetric difference. Returns all the items that are in *s* or *t*, but not in both sets.
`s.union(t)`	Union. Returns all items in *s* or *t*.

The `s.difference(t)`, `s.intersection(t)`, `s.symmetric_difference(t)`, and `s.union(t)` methods provide the standard mathematical operations on sets. The returned value has the same type as *s* (`set` or `frozenset`). The parameter *t* can be any Python object that supports iteration. This includes sets, lists, tuples, and strings. These set operations are also available as mathematical operators, as described further in Chapter 4.

Mutable sets (`set`) additionally provide the methods outlined in Table 3.8.

Table 3.8 **Methods for Mutable Set Types**

Item	Description
`s.add(item)`	Adds *item* to *s*. Has no effect if *item* is already in *s*.
`s.clear()`	Removes all items from *s*.
`s.difference_update(t)`	Removes all the items from *s* that are also in *t*.
`s.discard(item)`	Removes *item* from *s*. If *item* is not a member of *s*, nothing happens.

Table 3.8 **Continued**

Item	Description
s.intersection_update(*t*)	Computes the intersection of *s* and *t* and leaves the result in *s*.
s.pop()	Returns an arbitrary set element and removes it from *s*.
s.remove(*item*)	Removes *item* from *s*. If *item* is not a member, KeyError is raised.
s.symmetric_difference_update(t)	Computes the symmetric difference of *s* and *t* and leaves the result in *s*.
s.update(*t*)	Adds all the items in *t* to *s*. *t* may be another set, a sequence, or any object that supports iteration.

All these operations modify the set *s* in place. The parameter *t* can be any object that supports iteration.

Callable Types

Callable types represent objects that support the function call operation. There are several flavors of objects with this property, including user-defined functions, built-in functions, instance methods, and classes.

User-defined functions are callable objects created at the module level by using the def statement, at the class level by defining a static method, or with the lambda operator. Here's an example:

```
def foo(x,y):
    return x+y

class A(object):
    @staticmethod
    def foo(x,y):
        return x+y

bar = lambda x,y: x + y
```

A user-defined function *f* has the following attributes:

Attribute(s)	Description
f.__doc__ or *f*.func_doc	Documentation string
f.__name__ or *f*.func_name	Function name
f.__dict__ or *f*.func_dict	Dictionary containing function attributes
f.func_code	Byte-compiled code
f.func_defaults	Tuple containing the default arguments
f.func_globals	Dictionary defining the global namespace
f.func_closure	Tuple containing data related to nested scopes

Methods are functions that operate only on instances of an object. Two types of methods—instance methods and class methods—are defined inside a class definition, as shown here:

```
class Foo(object):
          def __init__(self):
            self.items = [ ]
          def update(self, x):
            self.items.append(x)
          @classmethod
          def whatami(cls):
            return cls
```

An instance method is a method that operates on an instance of an object. The instance is passed to the method as the first argument, which is called `self` by convention. Here's an example:

```
f = Foo()
f.update(2)       # update() method is applied to the object f
```

A class method operates on the class itself. The class object is passed to a class method in the first argument, `cls`. Here's an example:

```
Foo.whatami()   # Operates on the class Foo
f.whatami()     # Operates on the class of f (Foo)
```

A bound method object is a method that is associated with a specific object instance. Here's an example:

```
a = f.update        # a is a method bound to f
b = Foo.whatami     # b is method bound to Foo (classmethod)
```

In this example, the objects `a` and `b` can be called just like a function. When invoked, they will automatically apply to the underlying object to which they were bound. Here's an example:

```
a(4)        # Calls f.update(4)
b()         # Calls Foo.whatami()
```

Bound and unbound methods are no more than a thin wrapper around an ordinary function object. The following attributes are defined for method objects:

Attribute	**Description**
m.`__doc__`	Documentation string
m.`__name__`	Method name
m.`im_class`	Class in which this method was defined
m.`im_func`	Function object implementing the method
m.`im_self`	Instance associated with the method (`None` if unbound)

So far, this discussion has focused on functions and methods, but class objects (described shortly) are also callable. When a class is called, a new class instance is created. In addition, if the class defines an `__init__()` method, it's called to initialize the newly created instance.

An object instance is also callable if it defines a special method, `__call__()`. If this method is defined for an instance, *x*, then `x(args)` invokes the method `x.__call__(args)`.

The final types of callable objects are built-in functions and methods, which correspond to code written in extension modules and are usually written in C or C++. The following attributes are available for built-in methods:

Attribute	Description
b.`__doc__`	Documentation string
b.`__name__`	Function/method name
b.`__self__`	Instance associated with the method

For built-in functions such as `len()`, `__self__` is set to `None`, indicating that the function isn't bound to any specific object. For built-in methods such as *x*`.append()`, where *x* is a list object, `__self__` is set to *x*.

Finally, it is important to note that all functions and methods are first-class objects in Python. That is, function and method objects can be freely used like any other type. For example, they can be passed as arguments, placed in lists and dictionaries, and so forth.

Classes and Types

When you define a class, the class definition normally produces an object of type `type`. Here's an example:

```
>>> class Foo(object):
...     pass
...
>>> type(Foo)
<type 'type'>
```

When an object instance is created, the type of the instance is the class that defined it. Here's an example:

```
>>> f = Foo()
>>> type(f)
<class '__main__.Foo'>
```

More details about the object-oriented interface can be found in Chapter 7. However, there are a few attributes of types and instances that may be useful. If *t* is a type or class, then the attribute *t*.`__name__` contains the name of the type. The attributes *t*.`__bases__` contains a tuple of base classes. If *o* is an object instance, the attribute *o*.`__class__` contains a reference to its corresponding class and the attribute *o*.`__dict__` is a dictionary used to hold the object's attributes.

Modules

The *module* type is a container that holds objects loaded with the `import` statement. When the statement `import foo` appears in a program, for example, the name `foo` is assigned to the corresponding module object. Modules define a namespace that's implemented using a dictionary accessible in the attribute `__dict__`. Whenever an attribute of a module is referenced (using the dot operator), it's translated into a dictionary lookup. For example, *m*.*x* is equivalent to *m*.`__dict__["`*x*`"]`. Likewise, assignment to an attribute such as *m*.*x* = *y* is equivalent to *m*.`__dict__["`*x*`"] =` *y*. The following attributes are available:

Attribute	Description
m.`__dict__`	Dictionary associated with the module
m.`__doc__`	Module documentation string
m.`__name__`	Name of the module
m.`__file__`	File from which the module was loaded
m.`__path__`	Fully qualified package name, defined when the module object refers to a package

Files

The file object represents an open file and is returned by the built-in `open()` function (as well as a number of functions in the standard library). The methods on this type include common I/O operations such as `read()` and `write()`. However, because I/O is covered in detail in Chapter 9, readers should consult that chapter for more details.

Internal Types

A number of objects used by the interpreter are exposed to the user. These include traceback objects, code objects, frame objects, generator objects, slice objects, and the Ellipsis object. It is rarely necessary to manipulate these objects directly. However, their attributes are provided in the following sections for completeness.

Code Objects

Code objects represent raw byte-compiled executable code, or bytecode, and are typically returned by the built-in `compile()` function. Code objects are similar to functions except that they don't contain any context related to the namespace in which the code was defined, nor do code objects store information about default argument values. A code object, *c*, has the following read-only attributes:

Attribute	Description
c.`co_name`	Function name.
c.`co_argcount`	Number of positional arguments (including default values).
c.`co_nlocals`	Number of local variables used by the function.
c.`co_varnames`	Tuple containing names of local variables.
c.`co_cellvars`	Tuple containing names of variables referenced by nested functions.
c.`co_freevars`	Tuple containing names of free variables used by nested functions.
c.`co_code`	String representing raw bytecode.
c.`co_consts`	Tuple containing the literals used by the bytecode.
c.`co_names`	Tuple containing names used by the bytecode.
c.`co_filename`	Name of the file in which the code was compiled.
c.`co_firstlineno`	First line number of the function.
c.`co_lnotab`	String encoding bytecode offsets to line numbers.

Attribute	Description
c.co_stacksize	Required stack size (including local variables).
c.co_flags	Integer containing interpreter flags. Bit 2 is set if the function uses a variable number of positional arguments using "*args". Bit 3 is set if the function allows arbitrary keyword arguments using "**kwargs". All other bits are reserved.

Frame Objects

Frame objects are used to represent execution frames and most frequently occur in traceback objects (described next). A frame object, *f*, has the following read-only attributes:

Attribute	Description
f.f_back	Previous stack frame (toward the caller).
f.f_code	Code object being executed.
f.f_locals	Dictionary used for local variables.
f.f_globals	Dictionary used for global variables.
f.f_builtins	Dictionary used for built-in names.
f.f_restricted	Set to 1 if executing in restricted execution mode.
f.f_lineno	Line number.
f.f_lasti	Current instruction. This is an index into the bytecode string of f_code.

The following attributes can be modified (and are used by debuggers and other tools):

Attribute	Description
f.f_trace	Function called at the start of each source code line
f.f_exc_type	Most recent exception type
f.f_exc_value	Most recent exception value
f.f_exc_traceback	Most recent exception traceback

Traceback Objects

Traceback objects are created when an exception occurs and contains stack trace information. When an exception handler is entered, the stack trace can be retrieved using the sys.exc_info() function. The following read-only attributes are available in traceback objects:

Attribute	Description
t.tb_next	Next level in the stack trace (toward the execution frame where the exception occurred)
t.tb_frame	Execution frame object of the current level
t.tb_line	Line number where the exception occurred
t.tb_lasti	Instruction being executed in the current level

Generator Objects

Generator objects are created when a generator function is invoked (see Chapter 6, "Functions and Functional Programming"). A generator function is defined whenever a function makes use of the special `yield` keyword. The generator object serves as both an iterator and a container for information about the generator function itself. The following attributes and methods are available:

Attribute	Description
g.`gi_frame`	Execution frame of the generator function.
g.`gi_running`	Integer indicating whether or not the generator function is currently running.
g.`next()`	Execute the function until the next yield statement and return the value.

Slice Objects

Slice objects are used to represent slices given in extended slice syntax, such as `a[i:j:stride]`, `a[i:j, n:m]`, or `a[..., i:j]`. Slice objects are also created using the built-in `slice([i,] j [,stride])` function. The following read-only attributes are available:

Attribute	Description
s.`start`	Lower bound of the slice; `None` if omitted
s.`stop`	Upper bound of the slice; `None` if omitted
s.`step`	Stride of the slice; `None` if omitted

Slice objects also provide a single method, `s.indices(length)`. This function takes a length and returns a tuple `(start,stop,stride)` that indicates how the slice would be applied to a sequence of that length. For example:

```
s = slice(10,20)    #  Slice object represents [10:20]
s.indices(100)      #  Returns (10,20,1) --> [10:20]
s.indices(15)       #  Returns (10,15,1) --> [10:15]
```

Ellipsis Object

The Ellipsis object is used to indicate the presence of an ellipsis (...) in a slice. There is a single object of this type, accessed through the built-in name `Ellipsis`. It has no attributes and evaluates as `True`. None of Python's built-in types makes use of `Ellipsis`, but it may be used in third-party applications.

Classic Classes

In versions of Python prior to version 2.2, classes and objects were implemented using an entirely different mechanism that is now deprecated. For backward compatibility, however, these classes, called *classic classes* or *old-style classes*, are still supported.

The reason that classic classes are deprecated is due to their interaction with the Python type system. Classic classes do not define new data types, nor is it possible to specialize any of the built-in types such as lists or dictionaries. To overcome this limitation, Python 2.2 unified types and classes while introducing a different implementation of user-defined classes.

A classic class is created whenever an object *does not* inherit (directly or indirectly) from `object`. For example:

```
# A modern class
class Foo(object):
     pass

# A classic class.  Note: Does not inherit from object
class Bar:
     pass
```

Classic classes are implemented using a dictionary that contains all the objects defined within the class and defines a namespace. References to class attributes such as `c.x` are translated into a dictionary lookup, `c.__dict__["x"]`. If an attribute isn't found in this dictionary, the search continues in the list of base classes. This search is depth first in the order that base classes were specified in the class definition. An attribute assignment such as `c.y = 5` always updates the `__dict__` attribute of `c`, not the dictionaries of any base class.

The following attributes are defined by class objects:

Attribute	**Description**
`c.__dict__`	Dictionary associated with the class
`c.__doc__`	Class documentation string
`c.__name__`	Name of the class
`c.__module__`	Name of the module in which the class was defined
`c.__bases__`	Tuple containing base classes

A *class instance* is an object created by calling a class object. Each instance has its own local namespace that's implemented as a dictionary. This dictionary and the associated class object have the following attributes:

Attribute	**Description**
`x.__dict__`	Dictionary associated with an instance
`x.__class__`	Class to which an instance belongs

When the attribute of an object is referenced, such as in `x.a`, the interpreter first searches in the local dictionary for `x.__dict__["a"]`. If it doesn't find the name locally, the search continues by performing a lookup on the class defined in the `__class__` attribute. If no match is found, the search continues with base classes, as described earlier. If still no match is found and the object's class defines a `__getattr__()` method, it's used to perform the lookup. The assignment of attributes such as `x.a = 4` always updates `x.__dict__`, not the dictionaries of classes or base classes.

Special Methods

All the built-in data types implement a collection of special object methods. The names of special methods are always preceded and followed by double underscores (`__`). These methods are automatically triggered by the interpreter as a program executes. For example, the operation `x + y` is mapped to an internal method, `x.__add__(y)`, and an indexing operation, `x[k]`, is mapped to `x.__getitem__(k)`. The behavior of each data type depends entirely on the set of special methods that it implements.

User-defined classes can define new objects that behave like the built-in types simply by supplying an appropriate subset of the special methods described in this section. In addition, built-in types such as lists and dictionaries can be specialized (via inheritance) by redefining some of the special methods.

Object Creation, Destruction, and Representation

The methods in Table 3.9 create, initialize, destroy, and represent objects. `__new__()` is a static method that is called to create an instance (although this method is rarely redefined). The `__init__()` method initializes the attributes of an object and is called immediately after an object has been newly created. The `__del__()` method is invoked when an object is about to be destroyed. This method is invoked only when an object is no longer in use. It's important to note that the statement `del x` only decrements an object's reference count and doesn't necessarily result in a call to this function. Further details about these methods can be found in Chapter 7.

Table 3.9 **Special Methods for Object Creation, Destruction, and Representation**

Method	Description
`__new__(cls [,*args [,**kwargs]])`	A static method called to create a new instance
`__init__(self [,*args [,**kwargs]])`	Called to initialize a new instance
`__del__(self)`	Called to destroy an instance
`__repr__(self)`	Creates a full string representation of an object
`__str__(self)`	Creates an informal string representation
`__cmp__(self,other)`	Compares two objects and returns negative, zero, or positive
`__hash__(self)`	Computes a 32-bit hash index
`__nonzero__(self)`	Returns `0` or `1` for truth-value testing
`__unicode__(self)`	Creates a Unicode string representation

The `__new__()` and `__init__()` methods are used to create and initialize new instances. When an object is created by calling `A(args)`, it is translated into the following steps:

```
x = A.__new__(A,args)
is isinstance(x,A): x.__init__(args)
```

The `__repr__()` and `__str__()` methods create string representations of an object. The `__repr__()` method normally returns an expression string that can be evaluated to re-create the object. This method is invoked by the built-in `repr()` function and by the backquotes operator (`` ` ``). For example:

```
a = [2,3,4,5]       # Create a list
s = repr(a)         # s = '[2, 3, 4, 5]'
                    # Note : could have also used s = `a`
b = eval(s)         # Turns s back into a list
```

If a string expression cannot be created, the convention is for `__repr__()` to return a string of the form `<...`*`message`*`...>`, as shown here:

```
f = open("foo")
a = repr(f)       # a = "<open file 'foo', mode 'r' at dc030>"
```

The `__str__()` method is called by the built-in `str()` function and by the `print` statement. It differs from `__repr__()` in that the string it returns can be more concise and informative to the user. If this method is undefined, the `__repr__()` method is invoked.

The `__cmp__(`*`self`*`,`*`other`*`)` method is used by all the comparison operators. It returns a negative number if *`self`* `<` *`other`*, zero if *`self`* `==` *`other`*, and positive if *`self`* `>` *`other`*. If this method is undefined for an object, the object will be compared by object identity. In addition, an object may define an alternative set of comparison functions for each of the relational operators. These are known as rich comparisons and are described shortly. The `__nonzero__()` method is used for truth-value testing and should return `0` or `1` (or `True` or `False`). If undefined, the `__len__()` method is invoked to determine truth.

Finally, the `__hash__()` method computes an integer hash key used in dictionary operations (the hash value can also be returned using the built-in function `hash()`). The value returned should be identical for two objects that compare as equal. Furthermore, mutable objects should not define this method; any changes to an object will alter the hash value and make it impossible to locate an object on subsequent dictionary lookups. An object should not define a `__hash__()` method without also defining `__cmp__()`.

Attribute Access

The methods in Table 3.10 read, write, and delete the attributes of an object using the dot (`.`) operator and the `del` operator, respectively.

Table 3.10 **Special Methods for Attribute Access**

Method	**Description**
`__getattribute__(`*`self`*`,`*`name`*`)`	Returns the attribute *`self.name`*.
`__getattr__(`*`self`*`, `*`name`*`)`	Returns the attribute *`self.name`* if not found through normal attribute lookup.
`__setattr__(`*`self`*`, `*`name`*`, `*`value`*`)`	Sets the attribute *`self.name`* `=` *`value`*. Overrides the default mechanism.
`__delattr__(`*`self`*`, `*`name`*`)`	Deletes the attribute *`self.name`*.

An example will illustrate:

```
class Foo(object):
     def __init__(self):
           self.x = 37

f = Foo()

a = f.x        # Invokes __getattribute__(f,"x")
b = f.y        # Invokes __getattribute__(f,"y") --> Not found
               # Then invokes __getattr__(f,"y")
```

```
f.x = 42       # Invokes __setattr__(f,"x",42)
f.y = 93       # Invokes __setattr__(f,"y",93)

del f.y        # Invokes __delattr__(f,"y")
```

Whenever an attribute is accessed, the `__getattribute__()` method is always invoked. If the attribute is located, it is returned. Otherwise, the `__getattr__()` method is invoked. The default behavior of `__getattr__()` is to raise an `AttributeError` exception. The `__setattr__()` method is always invoked when setting an attribute, and the `__delattr__()` method is always invoked when deleting an attribute.

A subtle aspect of attribute access concerns a special kind of attribute known as a *descriptor*. A descriptor is an object that implements one or more of the methods in Table 3.11.

Table 3.11 **Special Methods for Descriptor Attributes**

Method	Description
`__get__(`*`self`*`,`*`instance`*`,`*`owner`*`)`	Returns an attribute value or raises `AttributeError`
`__set__(`*`self`*`,`*`instance`*`,`*`value`*`)`	Sets the attribute to *`value`*
`__delete__(`*`self`*`,`*`instance`*`)`	Deletes the attribute

Essentially, a descriptor attribute knows how to compute, set, and delete its own value whenever it is accessed. Typically, it is used to provide advanced features of classes such as static methods and properties. For example:

```
class SimpleProperty(object):
    def __init__(self,fget,fset):
        self.fget = fget
        self.fset = fset
    def __get__(self,instance,cls):
        return self.fget(instance)       # Calls instance.fget()
    def __set__(self,instance,value)
        return self.fset(instance,value)  # Calls instance.fset(value)

class Circle(object):
     def __init__(self,radius):
          self.radius = radius
     def getArea(self):
          return math.pi*self.radius**2
     def setArea(self):
          self.radius = math.sqrt(area/math.pi)
     area = SimpleProperty(getArea,setArea)
```

In this example, the class `SimpleProperty` defines a descriptor in which two functions, *`fget`* and *`fset`*, are supplied by the user to get and set the value of an attribute (note that a more advanced version of this is already provided using the `property()` function described in Chapter 7). In the `Circle` class that follows, these functions are used to create a descriptor attribute called `area`. In subsequent code, the `area` attribute is accessed transparently.

```
c = Circle(10)
a = c.area          # Implicitly calls c.getArea()
c.area = 10.0       # Implicitly calls c.setArea(10.0)
```

Underneath the covers, access to the attribute `c.area` is being translated into an operation such as `Circle.__dict__['area'].__get__(c,Circle)`.

It is important to emphasize that descriptors can only be created at the class level. It is not legal to create descriptors on a per-instance basis by defining descriptor objects inside `__init__()` and other methods.

Sequence and Mapping Methods

The methods in Table 3.12 are used by objects that want to emulate sequence and mapping objects.

Table 3.12 **Methods for Sequences and Mappings**

Method	**Description**
`__len__(`*`self`*`)`	Returns the length of *`self`*
`__getitem__(`*`self, key`*`)`	Returns *`self`*`[`*`key`*`]`
`__setitem__(`*`self, key, value`*`)`	Sets *`self`*`[`*`key`*`] = `*`value`*
`__delitem__(`*`self, key`*`)`	Deletes *`self`*`[`*`key`*`]`
`__getslice__(`*`self,i,j`*`)`	Returns *`self`*`[`*`i:j`*`]`
`__setslice__(`*`self,i,j,s`*`)`	Sets *`self`*`[`*`i:j`*`] = `*`s`*
`__delslice__(`*`self,i,j`*`)`	Deletes *`self`*`[`*`i:j`*`]`
`__contains__(`*`self,obj`*`)`	Returns `True` if *`obj`* is in *`self`*; otherwise, returns False

Here's an example:

```
a = [1,2,3,4,5,6]
len(a)               # __len__(a)
x = a[2]             # __getitem__(a,2)
a[1] = 7             # __setitem__(a,1,7)
del a[2]             # __delitem__(a,2)
x = a[1:5]           # __getslice__(a,1,5)
a[1:3] = [10,11,12]  # __setslice__(a,1,3,[10,11,12])
del a[1:4]           # __delslice__(a,1,4)
```

The `__len__` method is called by the built-in `len()` function to return a nonnegative length. This function also determines truth values unless the `__nonzero__()` method has also been defined.

For manipulating individual items, the `__getitem__()` method can return an item by key value. The key can be any Python object, but is typically an integer for sequences. The `__setitem__()` method assigns a value to an element. The `__delitem__()` method is invoked whenever the `del` operation is applied to a single element.

The slicing methods support the slicing operator *`s`*`[`*`i:j`*`]`. The `__getslice__()` method returns a slice, which is normally the same type of sequence as the original object. The indices *`i`* and *`j`* must be integers, but their interpretation is up to the method. Missing values for *`i`* and *`j`* are replaced with `0` and `sys.maxint`, respectively. The `__setslice__()` method assigns values to a slice. Similarly, `__delslice__()` deletes all the elements in a slice.

The `__contains__()` method is used to implement the `in` operator.

In addition to implementing the methods just described, sequences and mappings implement a number of mathematical methods, including `__add__()`, `__radd__()`, `__mul__()`, and `__rmul__()` to support concatenation and sequence replication. These methods are described shortly.

Finally, Python supports an extended slicing operation that's useful for working with multidimensional data structures such as matrices and arrays. Syntactically, you specify an extended slice as follows:

```
a = m[0:100:10]          # Strided slice (stride=10)
b = m[1:10, 3:20]        # Multidimensional slice
c = m[0:100:10, 50:75:5] # Multiple dimensions with strides
m[0:5, 5:10] = n         # extended slice assignment
del m[:10, 15:]          # extended slice deletion
```

The general format for each dimension of an extended slice is `i:j[:stride]`, where *`stride`* is optional. As with ordinary slices, you can omit the starting or ending values for each part of a slice. In addition, a special object known as the `Ellipsis` and written as `...` is available to denote any number of trailing or leading dimensions in an extended slice:

```
a = m[..., 10:20]    # extended slice access with Ellipsis
m[10:20, ...] = n
```

When using extended slices, the `__getitem__()`, `__setitem__()`, and `__delitem__()` methods implement access, modification, and deletion, respectively. However, instead of an integer, the value passed to these methods is a tuple containing one or more slice objects and at most one instance of the `Ellipsis` type. For example,

```
a = m[0:10, 0:100:5, ...]
```

invokes `__getitem__()` as follows:

```
a = __getitem__(m, (slice(0,10,None), slice(0,100,5), Ellipsis))
```

Python strings, tuples, and lists currently provide some support for extended slices, which is described in Chapter 4. Special-purpose extensions to Python, especially those with a scientific flavor, may provide new types and objects with advanced support for extended slicing operations.

Iteration

If an object, *`obj`*, supports iteration, it must provide a method, *`obj`*`.__iter__()`, that returns an iterator object. The iterator object *`iter`*, in turn, must implement a single method, *`iter`*`.next()`, that returns the next object or raises `StopIteration` to signal the end of iteration. Both of these methods are used by the implementation of the `for` statement as well as other operations that implicitly perform iteration. For example, the statement `for x in s` is carried out by performing steps equivalent to the following:

```
_iter = s.__iter__()
while 1:
    try:
         x = _iter.next()
    except StopIteration:
         break
    # Do statements in body of for loop
    ...
```

Mathematical Operations

Table 3.13 lists special methods that objects must implement to emulate numbers. Mathematical operations are always evaluated from left to right; when an expression such as `x + y` appears, the interpreter tries to invoke the method `x.__add__(y)`. The special methods beginning with *r* support operations with reversed operands. These are invoked only if the left operand doesn't implement the specified operation. For example, if *x* in `x + y` doesn't support the `__add__()` method, the interpreter tries to invoke the method `y.__radd__(x)`.

Table 3.13 **Methods for Mathematical Operations**

Method	Result
`__add__(self,other)`	`self + other`
`__sub__(self,other)`	`self - other`
`__mul__(self,other)`	`self * other`
`__div__(self,other)`	`self / other`
`__truediv__(self,other)`	`self / other` (future)
`__floordiv__(self,other)`	`self // other`
`__mod__(self,other)`	`self % other`
`__divmod__(self,other)`	`divmod(self,other)`
`__pow__(self,other [,modulo])`	`self ** other`, `pow(self, other, modulo)`
`__lshift__(self,other)`	`self << other`
`__rshift__(self,other)`	`self >> other`
`__and__(self,other)`	`self & other`
`__or__(self,other)`	`self | other`
`__xor__(self,other)`	`self ^ other`
`__radd__(self,other)`	`other + self`
`__rsub__(self,other)`	`other - self`
`__rmul__(self,other)`	`other * self`
`__rdiv__(self,other)`	`other / self`
`__rtruediv__(self,other)`	`other / self` (future)
`__rfloordiv__(self,other)`	`other // self`
`__rmod__(self,other)`	`other % self`
`__rdivmod__(self,other)`	`divmod(other,self)`
`__rpow__(self,other)`	`other ** self`
`__rlshift__(self,other)`	`other << self`
`__rrshift__(self,other)`	`other >> self`
`__rand__(self,other)`	`other & self`
`__ror__(self,other)`	`other | self`
`__rxor__(self,other)`	`other ^ self`
`__iadd__(self,other)`	`self += other`

Table 3.13 **Continued**

Method	**Result**
`__isub__(self,other)`	`self -= other`
`__imul__(self,other)`	`self *= other`
`__idiv__(self,other)`	`self /= other`
`__itruediv__(self,other)`	`self /= other` (future)
`__ifloordiv__(self,other)`	`self //= other`
`__imod__(self,other)`	`self %= other`
`__ipow__(self,other)`	`self **= other`
`__iand__(self,other)`	`self &= other`
`__ior__(self,other)`	`self |= other`
`__ixor__(self,other)`	`self ^= other`
`__ilshift__(self,other)`	`self <<= other`
`__irshift__(self,other)`	`self >>= other`
`__neg__(self)`	`-self`
`__pos__(self)`	`+self`
`__abs__(self)`	`abs(self)`
`__invert__(self)`	`~self`
`__int__(self)`	`int(self)`
`__long__(self)`	`long(self)`
`__float__(self)`	`float(self)`
`__complex__(self)`	`complex(self)`
`__oct__(self)`	`oct(self)`
`__hex__(self)`	`hex(self)`
`__coerce__(self,other)`	`Type coercion`

The methods `__iadd__()`, `__isub__()`, and so forth are used to support in-place arithmetic operators such as `a+=b` and `a-=b` (also known as *augmented assignment*). A distinction is made between these operators and the standard arithmetic methods because the implementation of the in-place operators might be able to provide certain customizations such as performance optimizations. For instance, if the *`self`* parameter is not shared, it might be possible to modify its value in place without having to allocate a newly created object for the result.

The three flavors of division operators, `__div__()`, `__truediv__()`, and `__floordiv__()`, are used to implement true division (/) and truncating division (//) operations. The separation of division into two types of operators is a relatively recent change to Python that was started in Python 2.2, but which has far-reaching effects. As of this writing, the default behavior of Python is to map the `/` operator to `__div__()`. In the future, it will be remapped to `__truediv__()`. This latter behavior can currently be enabled as an optional feature by including the statement `from __future__ import division` in a program.

The conversion methods `__int__()`, `__long__()`, `__float__()`, and `__complex__()` convert an object into one of the four built-in numerical types. The `__oct__()` and `__hex__()` methods return strings representing the octal and hexadecimal values of an object, respectively.

The `__coerce__(x,y)` method is used in conjunction with mixed-mode numerical arithmetic. This method returns either a 2-tuple containing the values of *x* and *y* converted to a common numerical type, or `NotImplemented` (or `None`) if no such conversion is possible. To evaluate the operation *x op y*, where *op* is an operation such as `+`, the following rules are applied, in order:

1. If *x* has a `__coerce__()` method, replace *x* and *y* with the values returned by `x.__coerce__(y)`. If `None` is returned, skip to step 3.
2. If *x* has a method `__op__()`, return `x.__op__(y)`. Otherwise, restore *x* and *y* to their original values and continue.
3. If *y* has a `__coerce__()` method, replace *x* and *y* with the values returned by `y.__coerce__(x)`. If `None` is returned, raise an exception.
4. If *y* has a method `__rop__()`, return `y.__rop__(x)`. Otherwise, raise an exception.

Although strings define a few arithmetic operations, the `__coerce__()` method is not used in mixed-string operations involving standard and Unicode strings.

The interpreter supports only a limited number of mixed-type operations involving the built-in types, in particular the following:

- If *x* is a string, `x % y` invokes the string-formatting operation, regardless of the type of `y`.
- If *x* is a sequence, `x + y` invokes sequence concatenation.
- If either *x* or *y* is a sequence and the other operand is an integer, `x * y` invokes sequence repetition.

Comparison Operations

Table 3.14 lists special methods that objects can implement to provide individualized versions of the relational operators (`<`, `>`, `<=`, `>=`, `==`, `!=`). These are known as rich comparisons. Each of these functions takes two arguments and is allowed to return any kind of object, including a Boolean value, a list, or any other Python type. For instance, a numerical package might use this to perform an element-wise comparison of two matrices, returning a matrix with the results. If a comparison can't be made, these functions may also raise an exception.

Table 3.14 **Methods for Comparisons**

Method	Result
`__lt__(self,other)`	`self < other`
`__le__(self,other)`	`self <= other`
`__gt__(self,other)`	`self > other`
`__ge__(self,other)`	`self >= other`
`__eq__(self,other)`	`self == other`
`__ne__(self,other)`	`self != other`

Callable Objects

Finally, an object can emulate a function by providing the `__call__(self [,*args [, **kwargs]])` method. If an object, *x*, provides this method, it can be invoked like a function. That is, `x(arg1, arg2, ...)` invokes `x.__call__(self, arg1, arg2, ...)`.

Performance Considerations

The execution of a Python program is mostly a sequence of function calls involving the special methods described in the earlier section "Special Methods." If you find that a program runs slowly, you should first check to see if you're using the most efficient algorithm. After that, considerable performance gains can be made simply by understanding Python's object model and trying to eliminate the number of special method calls that occur during execution.

For example, you might try to minimize the number of name lookups on modules and classes. For example, consider the following code:

```
import math
d= 0.0
for i in xrange(1000000):
     d = d + math.sqrt(i)
```

In this case, each iteration of the loop involves two name lookups. First, the `math` module is located in the global namespace; then it's searched for a function object named `sqrt`. Now consider the following modification:

```
from math import sqrt
d = 0.0
for i in xrange(1000000):
     d = d + sqrt(i)
```

In this case, one name lookup is eliminated from the inner loop, resulting in a considerable speedup.

Unnecessary method calls can also be eliminated by making careful use of temporary values and avoiding unnecessary lookups in sequences and dictionaries. For example, consider the following two classes:

```
class Point(object):
    def __init__(self,x,y,z):
          self.x = x
          self.y = y
          self.z = z

class Poly(object):
    def __init__(self):
          self.pts = [ ]
    def addpoint(self,pt):
          self.pts.append(pt)
    def perimeter(self):
          d = 0.0
          self.pts.append(self.pts[0])     # Temporarily close the polygon
          for i in xrange(len(self.pts)-1):
               d2 = (self.pts[i+1].x - self.pts[i].x)**2 + \
                    (self.pts[i+1].y - self.pts[i].y)**2 + \
                    (self.pts[i+1].z - self.pts[i].z)**2
               d = d + math.sqrt(d2)
          self.pts.pop()                   # Restore original list of points
          return d
```

In the `perimeter()` method, each occurrence of `self.pts[i]` involves two special-method lookups—one involving a dictionary and another involving a sequence. You can reduce the number of lookups by rewriting the method as follows:

```
class Poly(object):
      ...
      def perimeter(self):
          d = 0.0
          pts = self.pts
          pts.append(pts[0])
          for i in xrange(len(pts)-1):
                p1 = pts[i+1]
                p2 = pts[i]
                d2 = (p1.x - p2.x)**2 + \
                     (p1.y - p2.y)**2 + \
                     (p1.z - p2.z)**2
                d = d + math.sqrt(d2)
          pts.pop()
          return d
```

Although the performance gains made by such modifications are often modest (15%–20%), an understanding of the underlying object model and the manner in which special methods are invoked can result in faster programs. Of course, if performance is extremely critical, you often can export functionality to a Python extension module written in C or C++.

This page intentionally left blank