Chapter 3. Python Basics

Chapter Topics

Our next goal is to go through the basic Python syntax, describe some general style guidelines, then brief you on identifiers, variables, and keywords. We will also discuss how memory space for variables is allocated and deallocated. Finally, we will be exposed to a much larger example Python programtaking the plunge, as it were. No need to worry, there are plenty of life preservers around that allow for swimming rather than the alternative.

3.1. Statements and Syntax

Some rules and certain symbols are used with regard to statements in Python:

- Hash mark (#) indicates Python comments
- NEWLINE (\n) is the standard line separator (one statement per line)
- Backslash (\ ) continues a line
- Semicolon (;) joins two statements on a line
- Colon (:) separates a header line from its suite
- Statements (code blocks) grouped as suites
- Suites delimited via indentation
- Python files organized as modules

3.1.1. Comments (#)

First things first: Although Python is one of the easiest languages to read, it does not preclude the programmer from proper and adequate usage and placement of comments in the code. Like many of its Unix scripting brethren, Python comment statements begin with the pound sign or hash symbol (#). A comment can begin anywhere on a line. All characters following the # to the end of the line are ignored by the interpreter. Use them wisely and judiciously.

3.1.2. Continuation (\ )

Python statements are, in general, delimited by NEWLINEs, meaning one statement per line. Single statements can be broken up into multiple lines by use of the backslash. The backslash symbol (\ ) can be placed before a NEWLINE to continue the current statement onto the next line.

```
# check conditions
if (weather_is_hot == 1) and \
   (shark_warnings == 0):
       send_goto_beach_mesg_to_pager()
```

There are two exceptions where lines can be continued without backslashes. A single statement can take up more than one line when enclosing operators are used, i.e., parentheses, square brackets, or braces, and when NEWLINEs are contained in strings enclosed in triple quotes.

```
# display a string with triple quotes
print '''hi there, this is a long message for you
that goes over multiple lines... you will find
out soon that triple quotes in Python allows
this kind of fun! it is like a day on the beach!'''

# set some variables
go_surf, get_a_tan_while, boat_size, toll_money = (1,
   'windsurfing', 40.0, -2.00)
```

Given a choice between using the backslash and grouping components you can break up with a NEWLINE, i.e., with parentheses, we recommend the latter as it is more readable.

3.1.3. Multiple Statement Groups as Suites (:)

Groups of individual statements making up a single code block are called "suites" in Python (as we introduced in Chapter 2). *Compound* or *complex statements*, such as **if**, **while**, **def**, and **class**, are those that require a header line and a suite. Header lines begin the statement (with the keyword) and terminate with a colon (tt) and are followed by one or more lines that make up the suite. We will refer to the combination of a header line and a suite as a *clause*.

3.1.4. Suites Delimited via Indentation

As we introduced in Section 2.10, Python employs indentation as a means of delimiting blocks of code. Code at inner levels are indented via spaces or tabs. Indentation requires exact indentation; in other words, all the lines of code in a suite must be indented at the exact same level (e.g., same number of spaces). Indented lines starting at different positions or column numbers are not allowed; each line would be considered part of another suite and would more than likely result in syntax errors.

Core Style: Indent with four spaces and avoid using tabs

As someone who is perhaps new to block delimitation using whitespace, a first obvious question might be: How many spaces should I use? *We think that two is too short, and six to eight is too many, so we suggest four spaces for everyone. Also, because tabs vary in the number of spaces depending on your system, we recommend not using tabs if there is any hint of cross-platform development. Both of these style guidelines are also supported by Guido van Rossum, the creator of Python, and documented in the Python Style Guide. You will find the same suggestions in our style guide in Section 3.4.*

A new code block is recognized when the amount of indentation has increased, an d its termination is signaled by a "dedentation," or a reduction of indentation matching a previous level's. Code that is not indented, i.e., the highest level of code, is considered the "main" portion of the script.

The decision to create code blocks in Python using indentation was based on the belief that grouping code in this manner is more elegant and contributes to the ease of reading to which we alluded earlier. It also helps avoid "dangling-else"-type problems, including ungrouped single statement clauses (those where a C **if** statement does not use braces at all, but has two indented statements following). The second statement will execute regardless of the conditional, leading to more programmer confusion until the light bulb finally blinks on.

Finally, no "holy brace wars" can occur when using indentation. In C (also C++ and Java), starting braces may be placed on the same line as the header statement, or may start the very next line, or may be indented on the next line. Some like it one way, some prefer the other, etc. You get the picture.

3.1.5. Multiple Statements on a Single Line (;)

The semicolon (;) allows multiple statements on a single line given that neither statement starts a new code block. Here is a sample snip using the semicolon:

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

We caution the reader to be wary of chaining multiple statements on individual lines as it makes code much less readable, thus less "Pythonic."

3.1.6. Modules

Each Python script is considered a *module*. Modules have a physical presence as disk files. When a module gets large enough or has diverse enough functionality, it may make sense to move some of the code out to another module. Code that resides in modules may belong to an application (i.e., a script that is directly executed), or may be executable code in a library-type module that may be "imported" from another module for invocation. As we mentioned in the last chapter, modules can contain blocks of code to run, class declarations, function declarations, or any combination of all of those.

PREV NEXT

3.2. Variable Assignment

This section focuses on variable assignment. We will discuss which identifiers make valid variables in Section 3.3.

Assignment Operator

The equal sign (=) is the main Python assignment operator. (The others are augmented assignment operator [see next section].)

```
anInt = -12
aString = 'cart'
aFloat = -3.1415 * (5.0 ** 2)
anotherString = 'shop' + 'ping'
aList = [3.14e10, '2nd elmt of a list', 8.82-4.371j]
```

Be aware now that assignment does not explicitly assign a value to a variable, although it may appear that way from your experience with other programming languages. In Python, objects are referenced, so on assignment, a reference (not a value) to an object is what is being assigned, whether the object was just created or was a pre-existing object. If this is not 100 percent clear now, do not worry about it. We will revisit this topic later on in the chapter, but just keep it in mind for now.

Also, if you are familiar with C, you know that assignments are treated as expressions. This is not the case in Python, where assignments do not have inherent values. Statements such as the following are invalid in Python:

```
>>> x = 1
>>> y = (x = x + 1)  # assignments not expressions!
  File "<stdin>", line 1
    y = (x = x + 1)
           ^
SyntaxError: invalid syntax
```

Chaining together assignments is okay, though (more on this later):

```
>>> y = x = x + 1
>>> x, y
(2, 2)
```

Augmented Assignment

2.0

Beginning in Python 2.0, the equal sign can be combined with an arithmetic operation and the resulting value reassigned to the existing variable. Known as *augmented assignment*, statements such as ...

```
x = x + 1
```

... can now be written as ...

```
x += 1
```

Augmented assignment refers to the use of operators, which imply both an arithmetic operation as well as an assignment. You will recognize the following symbols if you come from a C/C++ or Java background:

```
+=          -=        *=    /=     %=        **=
<<=   >>=         &=    ^= |=
```

Other than the obvious syntactical change, the most significant difference is that the first object (`A` in our example) is examined only once. Mutable objects will be modified in place, whereas immutable objects will have the same effect as `A = A + B` (with a new object allocated) except that `A` is only evaluated once, as we have mentioned before.

```
>>> m = 12
>>> m %= 7
>>> m
5
>>> m **= 2
>>> m
25
>>> aList = [123, 'xyz']
>>> aList += [45.6e7]
>>> aList
[123, 'xyz', 456000000.0]
```

Python does not support pre-/post-increment nor pre-/post-decrement operators such as `x++` or `--x`.

Multiple Assignment

```
>>> x = y = z = 1
>>> x
1
>>> y
1
>>> z
1
```

In the above example, an integer object (with the value 1) is created, and `x`, `y`, and `z` are all assigned the same reference to that object. This is the process of assigning a single object to multiple variables. It is also possible in Python to assign multiple objects to multiple variables.

"Multuple" Assignment

Another way of assigning multiple variables is using what we shall call the "multuple" assignment. This is not an official Python term, but we use "multuple" here because when assigning variables this way, the objects on both sides of the equal sign are tuples, a Python standard type we introduced in Section 2.8.

```
>>> x, y, z = 1, 2, 'a string'
>>> x
1
>>> y
2
>>> z
'a string'
```

In the above example, two integer objects (with values 1 and 2) and one string object are assigned to `x`, `y`, and `z` respectively. Parentheses are normally used to denote tuples, and although they are optional, we recommend them anywhere they make the code easier to read:

```
>>> (x, y, z) = (1, 2, 'a string')
```

If you have ever needed to swap values in other languages like C, you will be reminded that a temporary variable, i.e., `tmp`, is required to hold one value while the other is being exchanged:

```
/* swapping variables in C */
tmp = x;
x = y;
y = tmp;
```

In the above C code fragment, the values of the variables `x` and `y` are being exchanged. The `tmp` variable is needed to hold the value of one of the variables while the other is being copied into it. After that step, the original value kept in the temporary variable can be assigned to the second variable.

One interesting side effect of Python's "multuple" assignment is that we no longer need a temporary variable to swap the values of two variables.

```
# swapping variables in Python
>>> x, y = 1, 2
>>> x
1
>>> y
2
>>> x, y = y, x
>>> x
2
>>> y
1
```

Obviously, Python performs evaluation before making assignments.

3.3. Identifiers

Identifiers are the set of valid strings that are allowed as names in a computer language. From this all-encompassing list, we segregate out those that are *keywords*, names that form a construct of the language. Such identifiers are reserved words that may not be used for any other purpose, or else a syntax error (`SyntaxError` exception) will occur.

Python also has an additional set of identifiers known as *built-ins*, and although they are not reserved words, use of these special names is not recommended. (Also see Section 3.3.3.)

3.3.1. Valid Python Identifiers

The rules for Python identifier strings are like most other high-level programming languages that come from the C world:

- First character must be a letter or underscore (_)
- Any additional characters can be alphanumeric or underscore
- Case-sensitive

No identifiers can begin with a number, and no symbols other than the underscore are ever allowed. The easiest way to deal with underscores is to consider them as alphabetic characters. *Case-sensitivity* means that identifier `foo` is different from `Foo`, and both of those are different from `FOO`.

3.3.2. Keywords

Python's keywords are listed in Table 3.1. Generally, the keywords in any language should remain relatively stable, but should things ever change (as Python is a growing and evolving language), a list of keywords as well as an `iskeyword()` function are available in the `keyword` module.

Table 3.1. Python Keywords [a]

and	**as** [b]	**assert** [c]	**break**
class	**continue**	**def**	**del**
elif	**else**	**except**	**exec**
finally	**for**	**from**	**global**
if	**import**	**in**	**is**
lambda	**not**	**or**	**pass**
print	**raise**	**return**	**try**
while	**with** [b]	**yield** [d]	None [e]

[a] access keyword obsoleted as of Python 1.4.

[b] New in Python 2.6.

[c] New in Python 1.5.

[d] New in Python 2.3.

[e] Not a keyword but made a constant in Python 2.4.

3.3.3. Built-ins

In addition to keywords, Python has a set of "built-in" names available at any level of Python code that are either set and/or used by the interpreter. Although not keywords, built-ins should be treated as "reserved for the system" and not used for any other purpose. However, some circumstances may call for *overriding* (aka redefining, replacing) them. Python does not support overloading of identifiers, so only one name "binding" may exist at any given time.

We can also tell advanced readers that built-ins are members of the `__builtins__` module, which is automatically imported by the interpreter before your program begins or before you are given the >>> prompt in the interactive interpreter. Treat them like global variables that are available at any level of Python code.

3.3.4. Special Underscore Identifiers

Python designates (even more) special variables with underscores both prefixed and suffixed. We will also discover later that some are quite useful to the programmer while others are unknown or useless. Here is a summary of the special underscore usage in Python:

- `_xxx` Do not import with `'from module import *'`
- `__xxx__` System-defined name
- `__xxx` Request private name mangling in classes

Core Style: Avoid using underscores to begin variable names

Because of the underscore usage for special interpreter and built-in identifiers, we recommend that the programmer avoid beginning variable names with the underscore. Generally, a variable named `_xxx` is considered "private" and should not be used outside that module or class. It is good practice to use `_xxx` to denote when a variable is private. Since variables named `__xxx__` often mean special things to Python, you should avoid naming normal variables this way.

3.4. Basic Style Guidelines

Comments

You do not need to be reminded that comments are useful both to you and those who come after you. This is especially true for code that has been untouched by man (or woman) for a time (that means several months in software development time). Comments should not be absent, nor should there be novellas. Keep the comments explanatory, clear, short, and concise, but get them *in* there. In the end, it saves time and energy for everyone. Above all, make sure they stay accurate!

Documentation

Python also provides a mechanism whereby documentation strings can be retrieved dynamically through the `__doc__` special variable. The first unassigned string in a module, class declaration, or function declaration can be accessed using the attribute `obj.__doc__` where `obj` is the module, class, or function name. This works during runtime too!

Indentation

Since indentation plays a major role, you will have to decide on a spacing style that is easy to read as well as the least confusing. Common sense also plays a role in choosing how many spaces or columns to indent.

1 or 2	Probably not enough; difficult to determine which block of code statements belong to
8 to 10	May be too many; code that has many embedded levels will wrap around, causing the source to be difficult to read

Four spaces is very popular, not to mention being the preferred choice of Python's creator. Five and six are not bad, but text editors usually do *not* use these settings, so they are not as commonly used. Three and seven are borderline cases.

As far as tabs go, bear in mind that different text editors have different concepts of what tabs are. It is advised not to use tabs if your code will live and run on different systems or be accessed with different text editors.

Choosing Identifier Names

The concept of good judgment also applies in choosing logical identifier names. Decide on short yet meaningful identifiers for variables. Although variable length is no longer an issue with programming languages of today, it is still a good idea to keep name sizes reasonable length. The same applies for naming your modules (Python files).

Python Style Guide(s)

Guido van Rossum wrote up a Python Style Guide ages ago. It has since been replaced by no fewer than

three PEPs: 7 (Style Guide for C Code), 8 (Style Guide for Python Code), and 257 (DocString Conventions). These PEPs are archived, maintained, and updated regularly.

Over time, you will hear the term "Pythonic," which describes the Python way of writing code, organizing logic, and object behavior. Over more time, you will come to understand what that means. There is also another PEP, PEP 20, which lists the Zen of Python, starting you on your journey to discover what Pythonic really means. If you are not online and need to see this list, then use `import this` from your interpreter. Here are some links:

www.python.org/doc/essays/styleguide.html

www.python.org/dev/peps/pep-0007/

www.python.org/dev/peps/pep-0008/

www.python.org/dev/peps/pep-0020/

www.python.org/dev/peps/pep-0257/

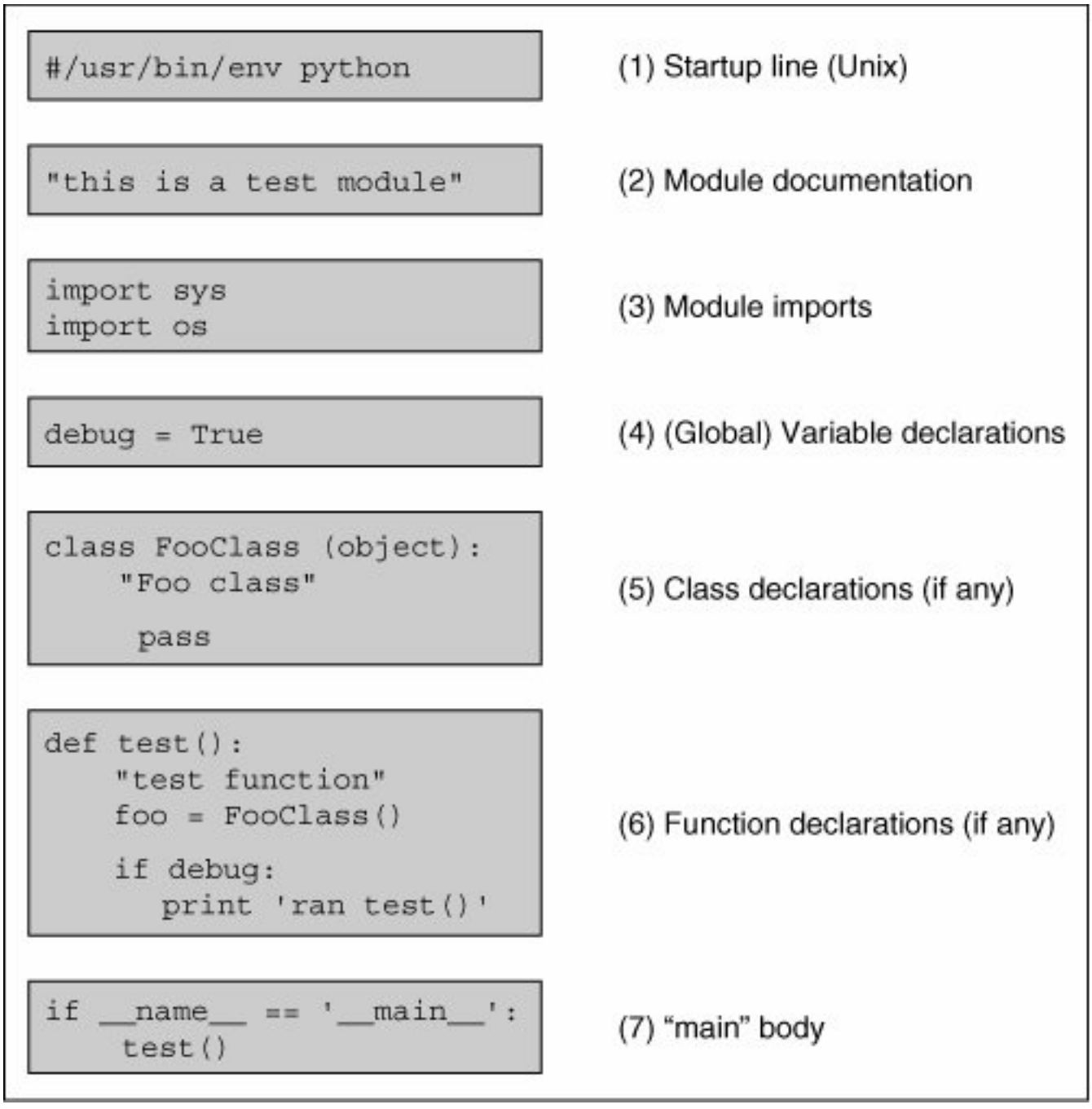
3.4.1. Module Structure and Layout

Modules are simply physical ways of logically organizing all your Python code. Within each file, you should set up a consistent and easy-to-read structure. One such layout is the following:

```
# (1) startup line (Unix)
# (2) module documentation
# (3) module imports
# (4) variable declarations
# (5) class declarations
# (6) function declarations
# (7) "main" body
```

Figure 3-1 illustrates the internal structure of a typical module.

Figure 3-1. Typical Python file structure



1. Startup line

   Generally used only in Unix environments, the startup line allows for script execution by name only (invoking the interpreter is not required).

2. Module documentation

   Summary of a module's functionality and significant global variables; accessible externally as *module*.__doc__.

3. Module imports

   Import all the modules necessary for all the code in current module; modules are imported once (when this module is loaded); imports within functions are not invoked until those functions are called.

4. Variable declarations

   Declare here (global) variables that are used by multiple functions in this module. We favor the use of local variables over globals, for good programming style mostly, and to a lesser extent, for improved performance and less memory usage.

5. Class declarations

   Any classes should be declared here. A class is defined when this module is imported and the **`class`** statement executed. Documentation variable is *`class.__doc__`*.

6. Function declarations

   Functions that are declared here are accessible externally as *`module.function()`*; function is defined when this module is imported and the **`def`** statement executed. Documentation variable is *`function`*`.__doc__`.

7. "main" body

   All code at this level is executed, whether this module is imported or started as a script; generally does not include much functional code, but rather gives direction depending on mode of execution.

Core Style: "main" calls `main( )`

The main body of code tends to contain lines such as the ones you see above, which check the `__name__` variable and take appropriate action (see Core Note on the following page). Code in the main body typically executes the class, function, and variable declarations, then checks `__name__` to see whether it should invoke another function (often called `main()`), which performs the primary duties of this module. The main body usually does no more than that. (Our example above uses `test()` rather than `main()` to avoid confusion until you read this Core Style sidebar.)

Regardless of the name, we want to emphasize that this is a great place to put a test suite in your code. As we explain in Section 3.4.2, most Python modules are created for import use only, and calling such a module directly should invoke a regression test of the code in such a module.

Most projects tend to consist of a single application and import any required modules. Thus it is important to bear in mind that most modules are created solely to be imported rather than to execute as scripts. We are more likely to create a Python library-style module whose sole purpose is to be imported by another module. After all, only one of the modulesthe one that houses the main applicationwill be executed, either by a user from the command line, by a batch or timed mechanism such as a Unix `cron` job, via a Web server call, or through a GUI callback.

With that fact in hand, we should also remember that all modules have the ability to execute code. All Python statements in the highest level of codethat is, the lines that are not indentedwill be executed on import, whether desired or not. Because of this "feature," safer code is written such that everything is in a function except for the code that should be executed on an import of a module. Again, usually only the main application module has the bulk of the executable code at its highest level. All other imported modules will have very little on the outside, and everything in functions or classes. (See Core Note that follows for more information.)

Core Note: `__name__` indicates how module was loaded

Because the "main" code is executed whether a module is imported or executed directly, we often need to know how this module was loaded to guide the execution path. An application may wish to import the module of another application, perhaps to access useful code which will otherwise have to be duplicated (not the OO thing to do). However, in this case, you only want access to this other application's code, not necessarily to run it. So the big question is, "Is there a way for Python to detect at runtime whether this module was imported or executed directly?" The answer is ... (drum roll ...) yes! The `__name__` system variable is the ticket.

- `__name__` *contains module name if imported*
- `__name__` *contains* `'__main__'` *if executed directly*

3.4.2. Create Tests in the Main Body

For good programmers and engineers, providing a test suite or harness for our entire application is the goal. Python simplifies this task particularly well for modules created solely for import. For these modules, you know that they would never be executed directly. Wouldn't it be nice if they were invoked to run code that puts that module through the test grinder? Would this be difficult to set up? Not really.

The test software should run only when this file is executed directly, i.e., not when it is imported from another module, which is the usual case. Above and in the Core Note, we described how we can determine whether a module was imported or executed directly. We can take advantage of this mechanism by using the `__name__` variable. If this module was called as a script, plug the test code right in there, perhaps as part of `main()` or `test()` (or whatever you decide to call your "second-level" piece of code) function, which is called only if this module is executed directly.

The "tester" application for our code should be kept current along with any new test criteria and results, and it should run as often as the code is updated. These steps will help improve the robustness of our code, not to mention validating and verifying any new features or updates.

Tests in the main body are an easy way to provide quick coverage of your code. The Python standard library also provides the `unittest` module, sometimes referred to as PyUnit, as a testing framework. Use of `unittest` is beyond the scope of this book, but it is something to consider when you need serious regression testing of a large system of components.

PREV NEXT

3.5. Memory Management

So far you have seen a large number of Python code samples. We are going to cover a few more details about variables and memory management in this section, including:

- Variables not declared ahead of time
- Variable types not declared
- No memory management on programmers' part
- Variable names can be "recycled"
- **del** statement allows for explicit "deallocation"

3.5.1. Variable Declarations (or Lack Thereof)

In most compiled languages, variables must be declared before they are used. In fact, C is even more restrictive: variables have to be declared at the beginning of a code block and before any statements are given. Other languages, like C++ and Java, allow "on-the-fly" declarations, i.e., those which occur in the middle of a body of codebut these name and type declarations are still required before the variables can be used. In Python, there are no explicit variable declarations. Variables are "declared" on first assignment. Like most languages, however, variables cannot be accessed until they are (created and) assigned:

```
>>> a
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: a
```

Once a variable has been assigned, you can access it by using its name:

```
>>> x = 4
>>> y = 'this is a string'
>>> x
4
>>> y
'this is a string'
```

3.5.2. Dynamic Typing

Another observation, in addition to lack of variable declaration, is the lack of type specification. In Python, the type and memory space for an object are determined and allocated at runtime. Although code is byte-compiled, Python is still an interpreted language. On creationthat is, on assignmentthe interpreter creates an object whose type is dictated by the syntax that is used for the operand on the right-hand side of an assignment. After the object is created, a reference to that object is assigned to the variable on the left-hand side of the assignment.

3.5.3. Memory Allocation

As responsible programmers, we are aware that when allocating memory space for variables, we are borrowing system resources, and eventually, we will have to return that which we borrowed back to the system. Python simplifies application writing because the complexities of memory management have been pushed down to the interpreter. The belief is that you should be using Python to solve problems with and not have to worry about lower-level issues that are not directly related to your solution.

3.5.4. Reference Counting

To keep track of objects in memory, Python uses the simple technique of *reference counting*. This means that internally, Python keeps track of all objects in use and how many interested parties there are for any particular object. You can think of it as simple as card-counting while playing the card game blackjack or 21. An internal tracking variable, called a *reference counter*, keeps track of how many references are being made to each object, called a *refcount* for short.

When an object is created, a reference is made to that object, and when it is no longer needed, i.e., when an object's refcount goes down to zero, it is garbage-collected. (This is not 100 percent true, but pretend it is for now.)

Incrementing the Reference Count

The refcount for an object is initially set to 1 when an object is created and (its reference) assigned.

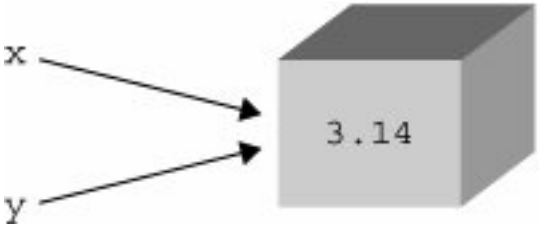
New references to objects, also called *aliases*, occur when additional variables are assigned to the same object, passed as arguments to invoke other bodies of code such as functions, methods, or class instantiation, or assigned as members of a sequence or mapping.

Let us say we make the following declarations:

```
x = 3.14
y = x
```

The statement `x = 3.14` allocates a floating point number (float) object and assigns a reference `x` to it. `x` is the first reference, hence setting that object's refcount to one. The statement `y = x` creates an alias `y`, which "points to" the same integer object as `x` (see Figure 3-2). A new object is *not* created for `y`.

Figure 3-2. An object with two references



Instead, the only thing that happens is that the reference count for this object is incremented by one (to 2). This is one way in which an object's refcount goes up. Other ways it can increment include the object being passed into a function call, and when the object is added to a container object such as a list.

In summary, an object's refcount is increased when

- It (the object) is created

```
x = 3.14
```

- Additional aliases for it are created

```
y = x
```

- It is passed to a function (new local reference)

```
foobar(x)
```

- It becomes part of a container object

```
myList = [123, x, 'xyz']
```

Now let us look at how reference counts go down.

Decrementing the Reference Count

When references to an object "go away," the refcount is decreased. The most obvious case is when a reference goes out of scope. This occurs most often when the function in which a reference is made completes. The local (automatic) variable is gone, and an object's reference counter is decremented.

A reference also goes away when a variable is reassigned to another object. For example:

```
foo = 'xyz'
bar = foo
foo = 123
```

The reference count for string object "`xyz`" is one when it is created and assigned to `foo`. It is then incremented when `bar` is added as an alias. However, when `foo` is reassigned to the integer 123, the reference count to "`xyz`" is decremented by one.

Other ways in which an object's reference count goes down include explicit removal of a reference using the **`del`** statement (see next section), when an object is removed from a container (or if the reference count to that container itself goes to zero).

In summary, an object's refcount is decreased when:

- A local reference goes out of scope, i.e., when `foobar()` (see previous example) terminates
- Aliases for that object are explicitly destroyed

```
del y            # or del x
```

- An alias is reassigned to another object (taking on a new reference)

```
x = 123
```

- It is explicitly removed from a container object

```
myList.remove(x)
```

- The container itself is deallocated

```
del myList        # or goes out-of-scope
```

See Section 11.8 for more information on variable scope.

`del` Statement

The `del` statement removes a single reference to an object. Its syntax is:

```
del obj1[, obj2[,... objN]]
```

For example, executing `del y` in the example above has two results:

- Removes name `y` from current namespace
- Lowers reference count to object `x` (by one)

Further still, executing `del x` will remove the final reference to the object, decrementing the reference counter to zero and causing the object to become "inaccessible" or "unreachable." It is at this point that the object becomes a candidate for garbage collection. Note that any tracing or debugging facility may keep additional references to an object, delaying or postponing that object from being garbage-collected.

3.5.5. Garbage Collection

Memory that is no longer being used is reclaimed by the system using a mechanism known as *garbage collection*. The interpreter keeps track of reference counts as above, but it is up to the garbage collector to deallocate the memory. The garbage collector is a separate piece of code that looks for objects with reference counts of zero. It is also responsible to check for objects with a reference count greater than zero that need to be deallocated. Certain situations lead to *cycles*.

A cyclic reference is where you have (at least two) objects that refer to each other, and even if all other references fall by the wayside, these references still exist, meaning that reference counting alone is not good enough.

Python's garbage collector is actually a combination of reference counting and the periodic invocation of a cyclic garbage collector. When an object's refcount reaches zero, the interpreter pauses to deallocate it and all objects that were reachable only from that object. In addition to this reference counting, the garbage collector also notices if a large number of objects have been allocated (and not deallocated though reference counting). In such cases, the interpreter will pause to try to clear out any unreferenced cycles.

3.6. First Python Programs

Now that we are familiar with the syntax, style, variable assignment, and memory allocation, it is time to look at slightly more complex code. You may or may not be familiar with all of the constructs of Python that we're going to show, but we believe that Python is so simple and elegant that you should be able to figure out what each piece of code does.

We are going to introduce two related scripts that manipulate text files. The first, `makeTextFile.py`, creates text files. It prompts the user for each line of text and writes the results to a file. The other, `readTextFile.py`, reads and displays the contents of a text file to the screen.

Take a look at both now, and see if you can figure out how each works.

Example 3.1. File Create (`makeTextFile.py`)

This application prompts the user for a (nonexistent) filename, then has the user enter each line of that file (one at a time). Finally, it writes the entire text file to disk.

```
1  #!/usr/bin/env python
2
3  'makeTextFile.py -- create text file'
4
5  import os
6  ls = os.linesep
7
8  # get filename
9  while True:
10
11     if os.path.exists(fname):
12        print "ERROR: '%s' already exists" % fname
13     else:
14        break
15
16 # get file content (text) lines
17 all = []
18 print "\nEnter lines ('.' by itself to quit).\n"
19
20 # loop until user terminates input
21 while True:
22     entry = raw_input('> ')
23     if entry == '.':
24         break
25     else:
26         all.append(entry)
27
28 # write lines to file with proper line-ending
29 fobj = open(fname, 'w')
30 fobj.writelines(['%s%s' % (x, ls) for x in all])
31 fobj.close()
32 print 'DONE!'
```

Lines 13

The Unix startup line is followed by the module documentation string. Keep your documentation string simple yet descriptive enough to be useful. Ours is a bit short, but so is this script. (We invite the reader to take a look at the documentation string at the commencement of the `cgi` module in the standard library for a seriously lengthy example of module documentation.)

Lines 56

We import the operating system (`os`) module next, and in line 6, we create a new local alias for the `linesep` attribute of that module. By doing this, we can shorten the name of the variable and also speed up access to it.

Core Tip: Use local variables to substitute for module attributes

Names like `os.linesep` require the interpreter to do two lookups: (1) `lookup os` to find that it is a module, and (2) look up the `linesep` attribute of that module. Because modules are also global variables, we pay another penalty. If you use an attribute like this often in a function, we recommend you alias it to a single local variable. Lookups are much fasterlocal variables are always searched first before globals, and we don't have attribute lookups either. This is one of the tricks in making your programs faster: replace often-used (and name-lengthy) module attributes with local references. Your code runs faster and has less clutter with a shorter name.

In our code snippet, we do not have a function to show you an example of using a local alias. Instead, we have a global alias, which is halfway there. At least we do not have to perform two lookups to get to the object.

Lines 814

If it is not apparent already, this is an "infinite loop," meaning we are presented with a body of code that will repeat and run forever unless we exit the looplook for a **`break`** statement somewhere! The **`while`** `true` conditional causes this to happen because **`while`** statements execute whenever its conditional expression evaluates to Boolean true, and `true` is Boolean true.

Lines 1014 prompt the user for an unused filename, meaning that the filename entered should not be the name of an already existing file. The `raw_input()` built-in function takes an argument to use as the prompt to the user. The resulting string entered by the user is the return value of `raw_input()`, which in this case gets assigned to `fname`.

If the user is unlucky enough to pick a name already in use, we notify the user and return the user to the prompt to enter another (file)name. Note that `os.path.exists()` is a helper function in the `os.path` (sub)module, which helps us make this determination. Only when a file with such a name does not exist,

meaning that `os.path.exists()` returns `False`, do we break out of this loop and continue.

Lines 1626

This is the part of our application that gives the user some instruction and prompts them for the contents of our text file, one line at a time. The `all` list will hold each linewe initialize it on line 17. Line 21 begins another infinite loop, which prompts the user for each line of the text file and only terminates when they enter a period '.' on a line by itself. The **`if-else`** statement on lines 2326 look for that sentinel and break out of the loop if it is seen (line 24); otherwise it adds another line to our total (line 26).

Lines 2832

Now that we have the entire contents in memory, we need to dump it to the text file. Line 29 opens the file for write, and line 30 writes each line to the file. Every file requires a line terminator (or termination character[s]). The construct on line 30, called a list comprehension, does the following: for every line in our file, append it with the appropriate line terminator for our platform. `'%s%s'` puts a line next to the termination character(s), and the grouping (`x`, `ls`) represents each line `x` of all lines and the terminatorfor Unix, it is `'\n'`, DOS and Win32, `'\r\n'`, etc. By using `os.linesep`, we do not need to have code to check which operating system this program is running on in order to determine which line terminating character(s) to use.

The file object's `writelines()` method then takes the resulting list of lines (now with terminators) and writes it to the file. The file is then closed in line 31, and we are *done*!

Not too bad, right? Now let us look at how to view the file we just created! For this, we have your second Python program, `readTextFile.py`. As you will see, it is much shorter than `makeTextfile.py`. The complexity of file creation is almost always greater than the reading of it. The only new and interesting part for you is the appearance of an exception handler.

Lines 13

These are the Unix startup line and module documentation string as usual.

Lines 57

Unlike `makeTextFile.py` where we kept pegging the user for names until they he or she chooses an unused filename, we don't care in this example.

Example 3.2. File Read and Display (`readTextFile.py`)

```
1  #!/usr/bin/env python
2
3  'readTextFile.py -- read and display text file'
4
5  # get filename
6  fname = raw_input('Enter filename: ')
7  print
8
9  # attempt to open file for reading
10 try:
11     fobj = open(fname, 'r')
12 except IOError, e:
13     print "*** file open error:", e
14 else:
15     # display contents to the screen
16     for eachLine in fobj:
17         print eachLine,
18     fobj.close()
```

In other words, we are performing the validation elsewhere (if at all). Line 7 just displays a new line to separate the prompting of the filename and the contents of the file.

Lines 918

This next Python construct (other than the comment) represents the rest of the script. This is a **`try-except-else`** statement. The **`try`** clause is a block of code that we want to monitor for errors. In our code (lines 1011), we are attempting to open the file with the name the user entered.

The **`except`** clause is where we decide what type of errors we're looking out for and what to do if such errors occur. In this case (lines 1213), we are checking to see if the file `open()` failedthis is usually an `IOError` type of error.

Finally, lines 1418 represent the **`else`** clause of a **`try-except`**the code that is executed if no errors occurred in the **`TRy`** block. In our case, we display each line of the file to the screen. Note that because we are not removing the trailing whitespace (line termination) characters from each line, we have to suppress the NEWLINE that the **`print`** statement automatically generatesthis is done by adding a trailing comma to the end of the **`print`** statement. We then close the file (line 18), which ends the program.

One final note regarding the use of `os.path.exists()` and an exception handler: The author is generally in favor of the former, when there is an existing function that can be used to detect error conditionsand even more simply, where the function is Boolean and gives you a "yes" or "no" answer. (Note that there is probably already an exception handler in such a function.) Why do you have to reinvent the wheel when there's already code just for that purpose?

An exception handler is best applied when there *isn't* such a convenient function, where you the programmer must recognize an "out of the ordinary" error condition and respond appropriately. In our case, we were able to dodge an exception because we check to see if a file exists, but there are many other situations that may cause a file open to fail, such as improper permissions, the connection to a network drive is out, etc. For safety's sake, you may end up with "checker" functions like `os.path.exists()` in addition to an exception handler, which may be able to take care of a situation where no such function is available.

You will find more examples of file system functions in Chapter 9 and more about exception handling in Chapter 10.

PREV NEXT

3.7. Related Modules/Developer Tools

The Python Style Guide (PEP 8), Python Quick Reference Guide, and the Python FAQ make for great reading as developer "tools." In addition, there are some modules that may help you become a more proficient Python programmer:

- Debugger: `pdb`
- Logger: `logging`
- Profilers: `profile`, `hotshot`, `cProfile`

The debugging module `pdb` allows you to set (conditional) breakpoints, single-step through lines of code, and check out stack frames. It also lets you perform post-mortem debugging.

The `logging` module, which was added in Python 2.3, defines functions and classes that help you implement a flexible logging system for your application. There are five levels of logging you can use: critical, error, warning, info, and debug.

Python has had a history of profilers, mostly because they were implemented at different times by different people with different needs. The original Python `profile` module was written in pure Python and measured the time spent in functions, the total time as well as the time spent per call, either only the time spent in particular functions or including subsequent (sub)functions calls from there. It is the oldest and the slowest of the three profilers but still gives useful profiling information.

The `hotshot` module was added in Python 2.2 and was intended to replace `profile` because it fixes various errors that `profile` was prone to and has improved performance due to being implemented in C. Note that `hotshot` focuses on reducing profiling overhead during execution but could take longer to deliver results. A critical bug in the timing code was fixed in Python 2.5.

The `cProfile` module, which was added in Python 2.5, was meant to replace the `hotshot` and `profile` modules. The one significant flaw identified by the authors of `cProfile` is that it takes a long time to load results from the log file, does not support detailed child function statistics, and some results appear inaccurate. It is also implemented in C.

3.8. Exercises

3-1. *Identifiers*. Why are variable name and type declarations not used in Python?

3-2. *Identifiers*. Why are function type declarations not used in Python?

3-3. *Identifiers*. Why should we avoid beginning and ending variable names with double underscores?

3-4. *Statements*. Can multiple Python statements be written on a single line?

3-5. *Statements*. Can a single Python statement be written over multiple lines?

3-6. *Variable Assignment*.

a.

Given the assignment `x, y, z = 1, 2, 3`, what do `x`, `y`, and `z` contain?

b.

What do `x`, `y`, and `z` contain after executing: `z, x, y = y, z, x`?

3-7. *Identifiers*. Which of the following are valid Python identifiers? If not, why not? Of the invalid ones, which are keywords?

```
int32          40XL        $aving$    printf      print
_print         this        self       __name__    0x40L
bool           true        big-daddy  2hot2touch  type
thisIsn'tAVar  thisIsAVar  R_U_Ready  Int         true
if             do          counter-1  access      _
```

The remaining problems deal with the `makeTextFile.py` and `readTextFile.py` programs.

3-8. *Python Code*. Copy the scripts to your file system and customize (tweak, improve) them. Modifications can include adding your own comments, changing the prompts ('`>`' is pretty boring), etc. Get comfortable looking at and editing Python code.

3-9. *Porting*. If you have Python installed on different types of computers, check to see if there are any differences in the `os.linesep` characters. Write down the type/OS and what `linesep` is.

3-10. *Exceptions*. Replace the call to `os.path.exists()` in `makeTextFile.py` with an exception handler as seen in `readTextFile.py`. On the flip side, replace the exception handler in `readTextFile.py` with a call to `os.path.exists()`.

3-11. *String Formatting*. Rather than suppressing the NEWLINE character generated by the **`print`** statement in `readTextFile.py`, change your code so that you strip each line of its whitespace before displaying it. In this case, you can remove the trailing comma from the **`print`** statement. Hint: Use the string `strip()` method.

3-12. *Merging Source Files*. Combine both programs into onecall it anything you like, perhaps `readNwriteTextFiles.py`. Let the user choose whether to create or display a text file.

3-13. **Adding Features*. Take your `readNwriteTextFiles.py` solution from the previous problem and add a major feature to it: Allow the user to edit an existing text file. You can do this any way you wish, whether you let the user edit line by line or the entire document at once. Note that the latter is much more difficult as you may need help from a GUI toolkit or a screen-based text editing module such as curses. Give users the option to apply the changes (saving the file) or discard them (leaving the original file intact), and also ensure the original file is preserved in case the program exits abnormally during operation.

PREV NEXT

Chapter 4. Python Objects

Chapter Topics

We will now begin our journey to the core part of the language. First we will introduce what Python objects are, then discuss the most commonly used built-in types. We then discuss the standard type operators and built-in functions (BIFs), followed by an insightful discussion of the different ways to categorize the standard types to gain a better understanding of how they work. Finally, we will conclude by describing some types that Python does *not* have (mostly as a benefit for those of you with experience in another high-level language).

4.1. Python Objects

Python uses the object model abstraction for data storage. Any construct that contains any type of value is an object. Although Python is classified as an "object-oriented programming (OOP) language," OOP is not required to create perfectly working Python applications. You can certainly write a useful Python script without the use of classes and instances. However, Python's object syntax and architecture encourage or "provoke" this type of behavior. Let us now take a closer look at what a Python object is.

All Python objects have the following three characteristics: an *identity*, a *type*, and a *value*.

IDENTITY	Unique identifier that differentiates an object from all others. Any object's identifier can be obtained using the `id()` built-in function (BIF). This value is as close as you will get to a "memory address" in Python (probably much to the relief of some of you). Even better is that you rarely, if ever, access this value, much less care what it is at all.
TYPE	An object's type indicates what kind of values an object can hold, what operations can be applied to such objects, and what behavioral rules these objects are subject to. You can use the `type()` BIF to reveal the type of a Python object. Since types are also objects in Python (did we mention that Python was object-oriented?), `type()` actually returns an object to you rather than a simple literal.
VALUE	Data item that is represented by an object.

All three are assigned on object creation and are read-only with one exception, the value. (For new-style types and classes, it may possible to change the type of an object, but this is not recommended for the beginner.) If an object supports updates, its value can be changed; otherwise, it is also read-only. Whether an object's value can be changed is known as an object's *mutability*, which we will investigate later on in Section 4.7. These characteristics exist as long as the object does and are reclaimed when an object is deallocated.

Python supports a set of basic (built-in) data types, as well as some auxiliary types that may come into play if your application requires them. Most applications generally use the standard types and create and instantiate classes for all specialized data storage.

4.1.1. Object Attributes

Certain Python objects have attributes, data values or executable code such as methods, associated with them. Attributes are accessed in the dotted attribute notation, which includes the name of the associated object, and were introduced in the Core Note in Section 2.14. The most familiar attributes are functions and methods, but some Python types have data attributes associated with them. Objects with data attributes include (but are not limited to): classes, class instances, modules, complex numbers, and files.

4.2. Standard Types

- Numbers (separate subtypes; three are integer types)
    - Integer
        - Boolean
        - Long integer
    - Floating point real number
    - Complex number
- String
- List
- Tuple
- Dictionary

We will also refer to standard types as "primitive data types" in this text because these types represent the primitive data types that Python provides. We will go over each one in detail in Chapters 5, 6, and 7.

4.3. Other Built-in Types

- Type
- Null object (`None`)
- File
- Set/Frozenset
- Function/Method
- Module
- Class

These are some of the other types you will interact with as you develop as a Python programmer. We will also cover all of these in other chapters of this book with the exception of the `type` and `None` types, which we will discuss here.

4.3.1. Type Objects and the `type` Type Object

It may seem unusual to regard types themselves as objects since we are attempting to just describe all of Python's types to you in this chapter. However, if you keep in mind that an object's set of inherent behaviors and characteristics (such as supported operators and built-in methods) must be defined somewhere, an object's type is a logical place for this information. The amount of information necessary to describe a type cannot fit into a single string; therefore types cannot simply be strings, nor should this information be stored with the data, so we are back to types as objects.

We will formally introduce the `type()` BIF below, but for now, we want to let you know that you can find out the type of an object by calling `type()` with that object:

```
>>> type(42)
<type 'int'>
```

Let us look at this example more carefully. It does not look tricky by any means, but examine the return value of the call. We get the seemingly innocent output of `<type 'int'>`, but what you need to realize is that this is not just a simple string telling you that 42 is an integer. What you see as `<type 'int'>` is actually a type object. It just so happens that the string representation chosen by its implementors has a string inside it to let you know that it is an `int` type object.

Now you may ask yourself, so then what is the type of any type object? Well, let us find out:

```
>>> type(type(42))
<type 'type'>
```

Yes, the type of all type objects is `type`. The `type` type object is also the mother of all types and is the default metaclass for all standard Python classes. It is perfectly fine if you do not understand this now. This will make sense as we learn more about classes and types.

With the unification of types and classes in Python 2.2, type objects are playing a more significant role in both object-oriented programming as well as day-to-day object usage. Classes are now types, and instances are now objects of their respective types.

4.3.2. `None`, Python's Null Object

Python has a special type known as the Null object or `NoneType`. It has only one value, `None`. The type of `None` is `NoneType`. It does not have any operators or BIFs. If you are familiar with C, the closest analogy to the `None` *type* is `void`, while the `None` *value* is similar to the C value of `NULL`. (Other similar objects and values include Perl's `undef` and Java's `void` type and `null` value.) `None` has no (useful) attributes and *always* evaluates to having a Boolean `False` value.

Core Note: Boolean values

All standard type objects can be tested for truth value and compared to objects of the same type. Objects have inherent `TRue` *or* `False` *values. Objects take a* `False` *value when they are empty, any numeric representation of zero, or the Null object* `None`.

The following are defined as having `false` *values in Python:*

- `None`
- `False` (*Boolean*)
- *Any numeric zero*:
- `0` (*integer*)
- `0.0` (*float*)
- `0L` (*long integer*)
- `0.0+0.0j` (*complex*)
- `""` (*empty string*)
- `[]` (*empty list*)
- `()` (*empty tuple*)
- `{}` (*empty dictionary*)

Any value for an object other than those above is considered to have a `true` *value, i.e., non-empty, non-zero, etc. User-created class instances have a* `false` *value when their nonzero* (`__nonzero__()`) *or length* (`__len__()`) *special methods, if defined, return a zero value.*

4.4. Internal Types

- Code
- Frame
- Traceback
- Slice
- Ellipsis
- Xrange

We will briefly introduce these internal types here. The general application programmer would typically not interact with these objects directly, but we include them here for completeness. Please refer to the source code or Python internal and online documentation for more information.

In case you were wondering about exceptions, they are now implemented as classes. In older versions of Python, exceptions were implemented as strings.

4.4.1. Code Objects

Code objects are executable pieces of Python source that are byte-compiled, usually as return values from calling the `compile()` BIF. Such objects are appropriate for execution by either **`exec`** or by the `eval()` BIF. All this will be discussed in greater detail in Chapter 14.

Code objects themselves do not contain any information regarding their execution environment, but they are at the heart of every user-defined function, all of which *do* contain some execution context. (The actual byte-compiled code as a code object is one attribute belonging to a function.) Along with the code object, a function's attributes also consist of the administrative support that a function requires, including its name, documentation string, default arguments, and global namespace.

4.4.2. Frame Objects

These are objects representing execution stack frames in Python. Frame objects contain all the information the Python interpreter needs to know during a runtime execution environment. Some of its attributes include a link to the previous stack frame, the code object (see above) that is being executed, dictionaries for the local and global namespaces, and the current instruction. Each function call results in a new frame object, and for each frame object, a C stack frame is created as well. One place where you can access a frame object is in a traceback object (see the following section).

4.4.3. Traceback Objects

When you make an error in Python, an exception is raised. If exceptions are not caught or "handled," the interpreter exits with some diagnostic information similar to the output shown below:

```
Traceback (innermost last):
  File "<stdin>", line N?, in ???
ErrorName: error reason
```

The traceback object is just a data item that holds the stack trace information for an exception and is created when an exception occurs. If a handler is provided for an exception, this handler is given access to the traceback object.

4.4.4. Slice Objects

Slice objects are created using the Python extended slice syntax. This extended syntax allows for different types of indexing. These various types of indexing include *stride indexing*, multi-dimensional indexing, and indexing using the Ellipsis type. The syntax for multi-dimensional indexing is *`sequence`*`[`*`start1`* : *`end1`*, *`start2 : end2]`*, or using the ellipsis, `sequence [..., start1 : end1]`. Slice objects can also be generated by the `slice()` BIF.

Stride indexing for sequence types allows for a third slice element that allows for "step"-like access with a syntax of *`sequence[starting_index : ending_index : stride]`*.

Support for the stride element of the extended slice syntax have been in Python for a long time, but until 2.3 was only available via the C API or Jython (and previously JPython). Here is an example of stride indexing:

```
>>> foostr = 'abcde'
>>> foostr[::-1]
'edcba'
>>> foostr[::-2]
'eca'
>>> foolist = [123, 'xba', 342.23, 'abc']
>>> foolist[::-1]
['abc', 342.23, 'xba', 123]
```

4.4.5. Ellipsis Objects

Ellipsis objects are used in extended slice notations as demonstrated above. These objects are used to represent the actual ellipses in the slice syntax (...). Like the Null object `None`, ellipsis objects also have a single name, `Ellipsis`, and have a Boolean `TRue` value at all times.

4.4.6. XRange Objects

XRange objects are created by the BIF `xrange()`, a sibling of the `range()` BIF, and used when memory is limited and when `range()` generates an unusually large data set. You can find out more about `range()` and `xrange()` in Chapter 8.

For an interesting side adventure into Python types, we invite the reader to take a look at the `types` module in the standard Python library.

4.5. Standard Type Operators

4.5.1. Object Value Comparison

2.3

Comparison operators are used to determine equality of two data values between members of the same type. These comparison operators are supported for all built-in types. Comparisons yield Boolean `TRue` or `False` values, based on the validity of the comparison expression. (If you are using Python prior to 2.3 when the Boolean type was introduced, you will see integer values 1 for `TRue` and 0 for `False`.) A list of Python's value comparison operators is given in Table 4.1.

Table 4.1. Standard Type Value Comparison Operators

Operator	*Function*
`expr1 < expr2`	*expr1* is less than *expr2*
`expr1 > expr2`	*expr1* is greater than *expr2*
`expr1 <= expr2`	*expr1* is less than or equal to *expr2*
`expr1 >= expr2`	*expr1* is greater than or equal to *expr2*
`expr1 == expr2`	*expr1* is equal to *expr2*
`expr1 != expr2`	*expr1* is not equal to *expr2* (C-style)
`expr1 <> expr2`	*expr1* is not equal to *expr2* (ABC/Pascal-style)[a]

[a] This "not equal" sign will be phased out in future version of Python. Use `!=` instead.

Note that comparisons performed are those that are appropriate for each data type. In other words, numeric types will be compared according to numeric value in sign and magnitude, strings will compare lexicographically, etc.

```
>>> 2 == 2
True
>>> 2.46 <= 8.33
True
>>> 5+4j >= 2-3j
True
>>> 'abc' == 'xyz'
False
```

```
>>> 'abc' > 'xyz'
False
>>> 'abc' < 'xyz'
True
>>> [3, 'abc'] == ['abc', 3]
False
>>> [3, 'abc'] == [3, 'abc']
True
```

Also, unlike many other languages, multiple comparisons can be made on the same line, evaluated in left-to-right order:

```
>>> 3 < 4 < 7           # same as ( 3 < 4 ) and ( 4 < 7 )
True
>>> 4 > 3 == 3         # same as ( 4 > 3 ) and ( 3 == 3 )
True
>>> 4 < 3 < 5 != 2 < 7
False
```

We would like to note here that comparisons are strictly between object values, meaning that the comparisons are between the data values and not the actual data objects themselves. For the latter, we will defer to the object identity comparison operators described next.

4.5.2. Object Identity Comparison

In addition to value comparisons, Python also supports the notion of directly *comparing objects* themselves. Objects can be assigned to other variables (by reference). Because each variable points to the same (shared) data object, any change effected through one variable will change the object and hence be reflected through all references to the same object.

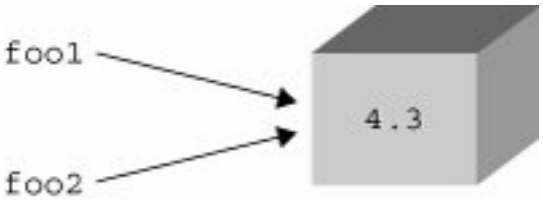
In order to understand this, you will have to think of variables as linking to objects now and be less concerned with the values themselves. Let us take a look at three examples.

Example 1: foo1 and foo2 reference the same object

```
foo1 = foo2 = 4.3
```

When you look at this statement from the value point of view, it appears that you are performing a multiple assignment and assigning the numeric value of 4.3 to both the `foo1` and `foo2` variables. This is true to a certain degree, but upon lifting the covers, you will find that a numeric object with the contents or value of 4.3 has been created. Then that object's reference is assigned to both `foo1` and `foo2`, resulting in both `foo1` and `foo2` aliased to the same object. Figure 4-1 shows an object with two references.

Figure 4-1. `foo1` and `foo2` reference the same object



Example 2: foo1 and foo2 reference the same object

```
foo1 = 4.3
foo2 = foo1
```

This example is very much like the first: A numeric object with value 4.3 is created, then assigned to one variable. When `foo2 = foo1` occurs, `foo2` is directed to the same object as `foo1` since Python deals with objects by passing references. `foo2` then becomes a new and additional reference for the original value. So both `foo1` and `foo2` now point to the same object. The same figure above applies here as well.

Example 3: foo1 and foo2 reference different objects

```
foo1 = 4.3
foo2 = 1.3 + 3.0
```

This example is different. First, a numeric object is created, then assigned to `foo1`. Then a second numeric object is created, and this time assigned to `foo2`. Although both objects are storing the exact same value, there are indeed two distinct objects in the system, with `foo1` pointing to the first, and `foo2` being a reference to the second. Figure 4-2 shows we now have two distinct objects even though both objects have the same value.

Figure 4-2. `foo1` and `foo2` reference different objects

Why did we choose to use boxes in our diagrams? Well, a good way to visualize this concept is to imagine a box (with contents inside) as an object. When a variable is assigned an object, that creates a "label" to stick on the box, indicating a reference has been made. Each time a new reference to the same object is made, another sticker is put on the box. When references are abandoned, then a label is removed. A box can be "recycled" only when all the labels have been peeled off the box. How does the system keep track of how many labels are on a box?

Each object has associated with it a counter that tracks the total number of references that exist to that object. This number simply indicates how many variables are "pointing to" any particular object. This is the *reference count* that we introduced in Chapter 3, Sections 3.5.5- 3.5.7 Python provides the **`is`** and **`is not`** operators to test if a pair of variables do indeed refer to the same object. Performing a check such as

```
a is b
```

is an equivalent expression to

```
id(a) == id(b)
```

The object identity comparison operators all share the same precedence level and are presented in Table 4.2.

Table 4.2. Standard Type Object Identity Comparison Operators

Operator	*Function*
obj1 **is** *obj2*	*obj1* is the same object as *obj2*
obj1 **is not** *obj2*	*obj1* is not the same object as *obj2*

In the example below, we create a variable, then another that points to the same object.

```
>>> a = [ 5, 'hat', -9.3]
>>> b = a
>>> a is b
True
>>> a is not b
False
>>>
>>> b = 2.5e-5
>>> b
2.5e-005
>>> a
[5, 'hat', -9.3]
>>> a is b
False
>>> a is not b
True
```

Both the **is** and **not** identifiers are Python keywords.

Core Note: Interning

In the above examples with the `foo1` and `foo2` objects, you will notice that we use floating point values rather than integers. The reason for this is although integers and strings are immutable objects, Python sometimes caches them to be more efficient. This would have caused the examples to appear that Python is not creating a new object when it should have. For example:

```
>>> a = 1
>>> id(a)
8402824
>>> b = 1
>>> id(b)
8402824
>>>
>>> c = 1.0
>>> id(c)
8651220
>>> d = 1.0
>>> id(d)
8651204
```

In the above example, `a` and `b` reference the same integer object, but `c` and `d` do not reference the same float object. If we were purists, we would want `a` and `b` to work just like `c` and `d` because we really did ask to create a new integer object rather than an alias, as in `b = a`.

Python caches or interns only simple integers that it believes will be used frequently in any Python application. At the time of this writing, Python interns integers in the `range(-1, 100)` but this is subject to change, so do not code your application to expect this.

In Python 2.3, the decision was made to no longer intern strings that do not have at least one reference outside of the "interned strings table." This means that without that reference, interned strings are no longer immortal and subject to garbage collection like everything else. A BIF introduced in 1.5 to request interning of strings, `intern()`, has now been deprecated as a result.

4.5.3. Boolean

Expressions may be linked together or negated using the Boolean logical operators **`and`**, **`or`**, and **`not`**, all of which are Python keywords. These Boolean operations are in highest-to-lowest order of precedence in Table 4.3. The **`not`** operator has the highest precedence and is immediately one level below all the comparison operators. The **`and`** and **`or`** operators follow, respectively.

Table 4.3. Standard Type Boolean Operators

Operator	*Function*
not *expr*	Logical NOT of *expr* (negation)
expr1 **and** *expr2*	Logical AND of *expr1* and *expr2* (conjunction)
expr1 **or***expr2*	Logical OR of *expr1* and *expr2* (disjunction)

```
>>> x, y = 3.1415926536, -1024
>>> x < 5.0
True
>>> not (x < 5.0)
False
>>> (x < 5.0) or (y > 2.718281828)
True
>>> (x < 5.0) and (y > 2.718281828)
False
>>> not (x is y)
True
```

Earlier, we introduced the notion that Python supports multiple comparisons within one expression. These expressions have an implicit **and** operator joining them together.

```
>>> 3 < 4 < 7    # same as "( 3 < 4 ) and ( 4 < 7 )"
True
```

PREV NEXT

4.6. Standard Type Built-in Functions

Along with generic operators, which we have just seen, Python also provides some BIFs that can be applied to all the basic object types: `cmp()`,`repr()`,`str()`,`type()`, and the single reverse or back quotes (``) operator, which is functionally equivalent to `repr()`.

Table 4.4. Standard Type Built-in Functions

Function	*Operation*
cmp(obj1, obj2)	Compares *obj1* and *obj2*, returns integer *i* where: *i* < 0 if *obj1* < *obj2* *i* > 0 if *obj1* > *obj2* *i* == 0 if *obj1* == *obj2*
repr(obj) or `obj`	Returns evaluatable string representation of *obj*
str(obj)	Returns printable string representation of *obj*
type(obj)	Determines type of *obj* and return type object

4.6.1. type()

We now formally introduce `type()`. In Python versions earlier than 2.2, `type()` is a BIF. Since that release, it has become a "factory function." We will discuss these later on in this chapter, but for now, you may continue to think of `type()` as a BIF. The syntax for `type()` is:

```
  type(object)
```

`type()` takes an object and returns its type. The return value is a `type` object.

```
  >>> type(4)                                    # int type
  <type 'int'>
  >>>
  >>> type('Hello World!')                     # string type
  <type 'string'>
  >>>
  >>> type(type(4))                              # type type
  <type 'type'>
```

In the examples above, we take an integer and a string and obtain their types using the `type()` BIF; in order to also verify that types themselves are types, we call `type()` on the output of a `type()` call.

Note the interesting output from the `type()` function. It does not look like a typical Python data type, i.e., a number or string, but is something enclosed by greater-than and less-than signs. This syntax is generally a clue that what you are looking at is an object. Objects may implement a printable string representation; however, this is not always the case. In these scenarios where there is no easy way to "display" an object, Python "pretty-prints" a string representation of the object. The format is usually of the form: *<object_something_or_another>*. Any object displayed in this manner generally gives the object type, an object ID or location, or other pertinent information.

4.6.2. `cmp()`

The `cmp()` BIF CoMPares two objects, say, *obj1* and *obj2*, and returns a negative number (integer) if *obj1* is less than *obj2*, a positive number if *obj1* is greater than *obj2*, and zero if *obj1* is equal to *obj2*. Notice the similarity in return values as C's `strcmp()`. The comparison used is the one that applies for that type of object, whether it be a standard type or a user-created class; if the latter, `cmp()` will call the class's special `__cmp__()` method. More on these special methods in Chapter 13, on Python classes. Here are some samples of using the `cmp()` BIF with numbers and strings.

```
>>> a, b = -4, 12
>>> cmp(a,b)
-1
>>> cmp(b,a)
1
>>> b = -4
>>> cmp(a,b)
0
>>>
>>> a, b = 'abc', 'xyz'
>>> cmp(a,b)
-23
>>> cmp(b,a)
23
>>> b = 'abc'
>>> cmp(a,b)
0
```

We will look at using `cmp()` with other objects later.

4.6.3. `str()` and `repr()` (and `` Operator)

The `str()` STRing and `repr()` REPResentation BIFs or the single back or reverse quote operator (``) come in very handy if the need arises to either re-create an object through evaluation or obtain a human-readable view of the contents of objects, data values, object types, etc. To use these operations, a Python object is provided as an argument and some type of string representation of that object is returned. In the examples that follow, we take some random Python types and convert them to their string representations.

```
>>> str(4.53-2j)
'(4.53-2j)'
>>>
>>> str(1)
'1'
>>>
```

```
>>> str(2e10)
'20000000000.0'
>>>
>>> str([0, 5, 9, 9])
'[0, 5, 9, 9]'
>>>
>>> repr([0, 5, 9, 9])
'[0, 5, 9, 9]'
>>>
>>> `[0, 5, 9, 9]`
'[0, 5, 9, 9]'
```

Although all three are similar in nature and functionality, only `repr()` and `` `` do exactly the same thing, and using them will deliver the "official" string representation of an object that can be evaluated as a valid Python expression (using the `eval()` BIF). In contrast, `str()` has the job of delivering a "printable" string representation of an object, which may not necessarily be acceptable by `eval()`, but will look nice in a **`print`** statement. There is a caveat that while most return values from `repr()` can be evaluated, not all can:

```
>>> eval(`type(type))`)
  File "<stdin>", line 1
    eval(`type(type))`)
                     ^
SyntaxError: invalid syntax
```

The executive summary is that `repr( )` is Python-friendly while `str()` produces human-friendly output. However, with that said, because both types of string representations coincide so often, on many occasions all three return the exact same string.

Core Note: Why have both `repr()` and `` `` ?

Occasionally in Python, you will find both an operator and a function that do exactly the same thing. One reason why both an operator and a function exist is that there are times where a function may be more useful than the operator, for example, when you are passing around executable objects like functions and where different functions may be called depending on the data item. Another example is the double-star (`**`) and `pow()` BIF, which performs "`x` to the `y` power" exponentiation for `x ** y` or `pow(x,y)`.

4.6.4. `type()` and `isinstance()`

Python does not support method or function overloading, so you are responsible for any "introspection" of the objects that your functions are called with. (Also see the Python FAQ 4.75.) Fortunately, we have the `type()` BIF to help us with just that, introduced earlier in Section 4.3.1.

What's in a name? Quite a lot, if it is the name of a type. It is often advantageous and/or necessary to base pending computation on the type of object that is received. Fortunately, Python provides a BIF just for that very purpose. `type()` returns the type for any Python object, not just the standard types. Using

the interactive interpreter, let us take a look at some examples of what `type()` returns when we give it various objects.

```
>>> type('')
<type 'str'>
>>>
>>> s = 'xyz'
>>> type(s)
<type 'str'>
>>>
>>> type(100)
<type 'int'>
>>> type(0+0j)
<type 'complex'>
>>> type(0L)
<type 'long'>
>>> type(0.0)
<type 'float'>
>>>
>>> type([])
<type 'list'>
>>> type(())
<type 'tuple'>
>>> type({})
<type 'dict'>
>>> type(type)
<type 'type'>
>>>
>>> class Foo: pass            # new-style class
...
>>> foo = Foo()
>>> class Bar(object): pass    # new-style class
...
>>> bar = Bar()
>>>
>>> type(Foo)
<type 'classobj'>
>>> type(foo)
<type 'instance'>
>>> type(Bar)
<type 'type'>
>>> type(bar)
<class '__main__.Bar'>
```

Types and classes were unified in Python 2.2. You will see output different from that above if you are using a version of Python older than 2.2:

```
>>> type('')
<type 'string'>
>>> type(0L)
<type 'long int'>
>>> type({})
<type 'dictionary'>
>>> type(type)
<type 'builtin_function_or_method'>
>>>
>>> type(Foo)          # assumes Foo created as in above
```

```
<type 'class'>
>>> type(foo)          # assumes foo instantiated also
<type 'instance'>
```

In addition to `type()`, there is another useful BIF called `isinstance()`. We cover it more formally in Chapter 13 (Object-Oriented Programming), but here we can introduce it to show you how you can use it to help determine the type of an object.

Example

We present a script in Example 4.1 that shows how we can use `isinstance()` and `type()` in a runtime environment. We follow with a discussion of the use of `type()` and how we migrated to using `isinstance()` instead for the bulk of the work in this example.

Example 4.1. Checking the Type (`typechk.py`)

The function `displayNumType()` takes a numeric argument and uses the `type()` built-in to indicate its type (or "not a number," if that is the case).

```
 1 #!/usr/bin/env python
 2
 3 def displayNumType(num):
 4     print num, 'is',
 5     if isinstance(num, (int, long, float, complex)):
 6        print 'a number of type:', type(num).__name__
 7     else:
 8       print 'not a number at all!!'
 9
10 displayNumType(-69)
11 displayNumType(99999999999999999999L)
12 displayNumType(98.6)
13 displayNumType(-5.2+1.9j)
14 displayNumType('xxx')
```

Running `typechk.py`, we get the following output:

```
-69 is a number of type: int
99999999999999999999 is a number of type: long
98.6 is a number of type: float
(-5.2+1.9j) is a number of type: complex
xxx is not a number at all!!
```

The Evolution of This Example

Original

The same function was defined quite differently in the first edition of this book:

```
def displayNumType(num):
    print num, "is",
    if type(num) == type(0):
        print 'an integer'
    elif type(num) == type(0L):
        print 'a long'
    elif type(num) == type(0.0):
        print 'a float'
    elif type(num) == type(0+0j):
        print 'a complex number'
    else:
        print 'not a number at all!!'
```

As Python evolved in its slow and simple way, so must we. Take a look at our original conditional expression:

```
if type(num) == type(0)...
```

Reducing Number of Function Calls

If we take a closer look at our code, we see a pair of calls to `type()`. As you know, we pay a small price each time a function is called, so if we can reduce that number, it will help with performance.

An alternative to comparing an object's type with a known object's type (as we did above and in the example below) is to utilize the `types` module, which we briefly mentioned earlier in the chapter. If we do that, then we can use the type object there without having to "calculate it." We can then change our code to only having one call to the `type()` function:

```
>>> import types
>>> if type(num) == types.IntType...
```

Object Value Comparison versus Object Identity Comparison

We discussed object value comparison versus object identity comparison earlier in this chapter, and if you realize one key fact, then it will become clear that our code is still not optimal in terms of performance. During runtime, there is always only one type object that represents an integer. In other words, `type(0)`, `type(42)`, `type(-100)` are always the same object: `<type 'int'>` (and this is also the same object as `types.IntType`).

If they are always the same object, then why do we have to compare their values since we already know they will be the same? We are "wasting time" extracting the values of both objects and comparing them if they are the same object, and it would be more optimal to just compare the objects themselves. Thus we have a migration of the code above to the following:

```
if type(num) is types.IntType... # or type(0)
```

Does that make sense? Object value comparison via the equal sign requires a comparison of their values, but we can bypass this check if the objects themselves are the same. If the objects are different,

then we do not even need to check because that means the original variable must be of a different type (since there is only one object of each type). One call like this may not make a difference, but if there are many similar lines of code throughout your application, then it starts to add up.

Reduce the Number of Lookups

This is a minor improvement to the previous example and really only makes a difference if your application performs makes many type comparisons like our example. To actually get the integer type object, the interpreter has to look up the `types` name first, and then within that module's dictionary, find `IntType`. By using **`from-import`**, you can take away one lookup:

```
from types import IntType
if type(num) is IntType ...
```

Convenience and Style

The unification of types and classes in 2.2 has resulted in the expected rise in the use of the `isinstance()` BIF. We formally introduce `isinstance()` in Chapter 13 (Object-Oriented Programming), but we will give you a quick preview now.

This Boolean function takes an object and one or more type objects and returns `true` if the object in question is an instance of one of the type objects. Since types and classes are now the same, `int` is now a type (object) and a class. We can use `isinstance()` with the built-in types to make our **`if`** statement more convenient and readable:

```
if isinstance(num, int)...
```

Using `isinstance()` along with type objects is now also the accepted style of usage when introspecting objects' types, which is how we finally arrive at our updated `typechk.py` application above. We also get the added bonus of `isinstance()` accepting a tuple of type objects to check against our object with instead of having an **`if-elif-else`** if we were to use only `type()`.

4.6.5. Python Type Operator and BIF Summary

A summary of operators and BIFs common to all basic Python types is given in Table 4.5. The progressing shaded groups indicate hierarchical precedence from highest-to-lowest order. Elements grouped with similar shading all have equal priority. Note that these (and most Python) operators are available as functions via the `operator` module.

Table 4.5. Standard Type Operators and Built-in Functions

Operator/Function	*Description*	*Result*[a]
String representation		

``	String representation	`str`
Built-in functions		
`cmp(obj1, obj2)`	Compares two objects	`int`
`repr(obj)`	String representation	`str`
`str(obj)`	String representation	`str`
`type(obj)`	Determines object type	`type`
Value comparisons		
<	Less than	`bool`
>	Greater than	`bool`
<=	Less than or equal to	`bool`
>=	Greater than or equal to	`bool`
==	Equal to	`bool`
!=	Not equal to	`bool`
<>	Not equal to	`bool`
Object comparisons		
`is`	The same as	`bool`
`is not`	Not the same as	`bool`
Boolean operators		
`not`	Logical negation	`bool`
`and`	Logical conjunction	`bool`
`or`	Logical disjunction	`bool`

[a] Boolean comparisons return either `TRue` or `False`.

4.7. Type Factory Functions

Since Python 2.2 with the unification of types and classes, all of the built-in types are now classes, and with that, all of the "conversion" built-in functions like `int()`, `type()`, `list()`, etc., are now *factory functions*. This means that although they look and act somewhat like functions, they are actually class names, and when you call one, you are actually instantiating an instance of that type, like a factory producing a good.

The following familiar factory functions were formerly built-in functions:

- `int(), long(), float(), complex()`
- `str(), unicode(), basestring()`
- `list(), tuple()`
- `type()`

Other types that did not have factory functions now do. In addition, factory functions have been added for completely new types that support the new-style classes. The following is a list of both types of factory functions:

- `dict()`
- `bool()`
- `set(), frozenset()`
- `object()`
- `classmethod()`
- `staticmethod()`
- `super()`
- `property()`
- `file()`

4.8. Categorizing the Standard Types

If we were to be maximally verbose in describing the standard types, we would probably call them something like Python's "basic built-in data object primitive types."

- "Basic," indicating that these are the standard or core types that Python provides
- "Built-in," due to the fact that these types come by default in Python
- "Data," because they are used for general data storage
- "Object," because objects are the default abstraction for data and functionality
- "Primitive," because these types provide the lowest-level granularity of data storage
- "Types," because that's what they are: data types!

However, this description does not really give you an idea of how each type works or what functionality applies to them. Indeed, some of them share certain characteristics, such as how they function, and others share commonality with regard to how their data values are accessed. We should also be interested in whether the data that some of these types hold can be updated and what kind of storage they provide.

There are three different models we have come up with to help categorize the standard types, with each model showing us the interrelationships between the types. These models help us obtain a better understanding of how the types are related, as well as how they work.

4.8.1. Storage Model

The first way we can categorize the types is by how many objects can be stored in an object of this type. Python's types, as well as types from most other languages, can hold either single or multiple values. A type which holds a single literal object we will call *atomic* or *scalar* storage, and those which can hold multiple objects we will refer to as *container* storage. (Container objects are also referred to as *composite* or *compound* objects in the documentation, but some of these refer to objects other than types, such as class instances.) Container types bring up the additional issue of whether different types of objects can be stored. All of Python's container types can hold objects of different types. Table 4.6 categorizes Python's types by storage model.

Table 4.6. Types Categorized by the Storage Model

Storage Model Category	*Python Types That Fit Category*
Scalar/atom	Numbers (all numeric types), strings (all are literals)
Container	Lists, tuples, dictionaries

Although strings may seem like a container type since they "contain" characters (and usually more than one character), they are not considered as such because Python does not have a character type (see Section 4.8). Thus strings are self-contained literals.

4.8.2. Update Model

Another way of categorizing the standard types is by asking the question, "Once created, can objects be changed, or can their values be updated?" When we introduced Python types early on, we indicated that certain types allow their values to be updated and others do not. *Mutable* objects are those whose values can be changed, and *immutable* objects are those whose values cannot be changed. Table 4.7 illustrates which types support updates and which do not.

Table 4.7. Types Categorized by the Update Model

Update Model Category	*Python Types That Fit Category*
Mutable	Lists, dictionaries
Immutable	Numbers, strings, tuples

Now after looking at the table, a thought that must immediately come to mind is, "Wait a minute! What do you mean that numbers and strings are immutable? I've done things like the following":

```
x = 'Python numbers and strings'
x = 'are immutable?!? What gives?'
i = 0
i = i + 1
```

"They sure as heck don't look immutable to me!" That is true to some degree, but looks can be deceiving. What is really happening behind the scenes is that the original objects are actually being replaced in the above examples. Yes, that is right. Read that again.

Rather than referring to the original objects, new objects with the new values were allocated and (re) assigned to the original variable names, and the old objects were garbage-collected. One can confirm this by using the `id()` BIF to compare object identities before and after such assignments.

If we added calls to `id()` in our example above, we may be able to see that the objects are being changed, as below:

```
>>> x = 'Python numbers and strings'
>>> print id(x)
16191392
>>> x = 'are immutable?!? What gives?'
>>> print id(x)
16191232
>>> i = 0
>>> print id(i)
7749552
>>> i = i + 1
>>> print id(i)
7749600
```

Your mileage will vary with regard to the object IDs as they will differ between executions. On the flip

side, lists can be modified without replacing the original object, as illustrated in the code below:

```
>>> aList = ['ammonia', 83, 85, 'lady']
>>> aList
['ammonia', 83, 85, 'lady']
>>>
>>> aList[2]
85
>>>
>>> id(aList)
135443480
>>>
>>> aList[2] = aList[2] + 1
>>> aList[3] = 'stereo'
>>> aList
['ammonia', 83, 86, 'stereo']
>>>
>>> id(aList)
135443480
>>>
>>> aList.append('gaudy')
>>> aList.append(aList[2] + 1)
>>> aList
['ammonia', 83, 86, 'stereo', 'gaudy', 87]
>>>
>>> id(aList)
135443480
```

Notice how for each change, the ID for the list remained the same.

4.8.3. Access Model

Although the previous two models of categorizing the types are useful when being introduced to Python, they are not the primary models for differentiating the types. For that purpose, we use the access model. By this, we mean, how do we access the values of our stored data? There are three categories under the access model: *direct*, *sequence*, and *mapping*. The different access models and which types fall into each respective category are given in Table 4.8.

Table 4.8. Types Categorized by the Access Model

Access Model Category	*Types That Fit Category*
Direct	Numbers
Sequence	Strings, lists, tuples
Mapping	Dictionaries

Direct types indicate single-element, non-container types. All numeric types fit into this category.

Sequence types are those whose elements are sequentially accessible via index values starting at 0. Accessed items can be either single elements or in groups, better known as slices. Types that fall into this category include strings, lists, and tuples. As we mentioned before, Python does not support a character type, so, although strings are literals, they are a sequence type because of the ability to access substrings sequentially.

Mapping types are similar to the indexing properties of sequences, except instead of indexing on a sequential numeric offset, elements (values) are unordered and accessed with a key, thus making mapping types a set of hashed key-value pairs.

We will use this primary model in the next chapter by presenting each access model type and what all types in that category have in common (such as operators and BIFs), then discussing each Python standard type that fits into those categories. Any operators, BIFs, and methods unique to a specific type will be highlighted in their respective sections.

So why this side trip to view the same data types from differing perspectives? Well, first of all, why categorize at all? Because of the high-level data structures that Python provides, we need to differentiate the "primitive" types from those that provide more functionality. Another reason is to be clear on what the expected behavior of a type should be. For example, if we minimize the number of times we ask ourselves, "What are the differences between lists and tuples again?" or "What types are immutable and which are not?" then we have done our job. And finally, certain categories have general characteristics that apply to all types in a certain category. A good craftsman (and craftswoman) should know what is available in his or her toolboxes.

The second part of our inquiry asks, "Why all these different models or perspectives"? It seems that there is no one way of classifying all of the data types. They all have crossed relationships with each other, and we feel it best to expose the different sets of relationships shared by all the types. We also want to show how each type is unique in its own right. No two types map the same across all categories. (Of course, all numeric subtypes do, so we are categorizing them together.) Finally, we believe that understanding all these relationships will ultimately play an important implicit role during development. The more you know about each type, the more you are apt to use the correct ones in the parts of your application where they are the most appropriate, and where you can maximize performance.

We summarize by presenting a cross-reference chart (see Table 4.9) that shows all the standard types, the three different models we use for categorization, and where each type fits into these models.

Table 4.9. Categorizing the Standard Types

Data Type	*Storage Model*	*Update Model*	*Access Model*
Numbers	Scalar	Immutable	Direct
Strings	Scalar	Immutable	Sequence
Lists	Container	Mutable	Sequence
Tuples	Container	Immutable	Sequence
Dictionaries	Container	Mutable	Mapping

4.9. Unsupported Types

Before we explore each standard type, we conclude this chapter by giving a list of types that are not supported by Python.

`char` or `byte`

Python does not have a char or byte type to hold either single character or 8-bit integers. Use strings of length one for characters and integers for 8-bit numbers.

pointer

Since Python manages memory for you, there is no need to access pointer addresses. The closest to an address that you can get in Python is by looking at an object's identity using the `id()` BIF. Since you have no control over this value, it's a moot point. However, under Python's covers, everything is a pointer.

`int` versus `short` versus `long`

Python's plain integers are the universal "standard" integer type, obviating the need for three different integer types, e.g., C's `int`, `short`, and `long`. For the record, Python's integers are implemented as C `long`s. Also, since there is a close relationship between Python's `int` and `long` types, users have even fewer things to worry about. You only need to use a single type, the Python integer. Even when the size of an integer is exceed, for example, multiplying two very large numbers, Python automatically gives you a long back instead of overflowing with an error.

`float` versus `double`

C has both a single precision `float` type and double-precision `double` type. Python's `float` type is actually a C `double`. Python does not support a single-precision floating point type because its benefits are outweighed by the overhead required to support two types of floating point types. For those wanting more accuracy and willing to give up a wider range of numbers, Python has a decimal floating point number too, but you have to import the `decimal` module to use the `Decimal` type. Floats are always estimations. Decimals are exact and arbitrary precision. Decimals make sense concerning things like money where the values are exact. Floats make sense for things that are estimates anyway, such as weights, lengths, and other measurements.

4.10. Exercises

4-1. *Python Objects*. What three attributes are associated with *all* Python objects? Briefly describe each one.

4-2. *Types*. What does immutable mean? Which Python types are mutable and which are not?

4-3. *Types*. Which Python types are sequences, and how do they differ from mapping types?

4-4. *`type()`*. What does the `type()` built-in function do? What kind of object does `type()` return?

4-5. *`str()` and `repr()`*. What are the differences between the `str()` *and* `repr()` built-in functions? Which is equivalent to the backquote (``) operator?

4-6. *Object Equality*. What do you think is the difference between the expressions `type(a) == type(b)` and `type(a) is type(b)`? Why is the latter preferred? What does `isinstance()` have to do it all of this?

4-7. `dir()` *Built-in Function*. In several exercises in Chapter 2, we experimented with a built-in function called `dir()`, which takes an object and reveals its attributes. Do the same thing for the `types` module. Write down the list of the types that you are familiar with, including all you know about each of these types; then create a separate list of those you are not familiar with. As you learn Python, deplete the "unknown" list so that all of them can be moved to the "familiar with" list.

4-8. *Lists and Tuples*. How are lists and tuples similar? Different?

4-9. **Interning*. Given the following assignments:

```
a = 10
b = 10
c = 100
d = 100
e = 10.0
f = 10.0
```

What is the output of each of the following and why?

a.

```
a is b
```

b.

```
c is d
```

c.

```
e is f
```

PREV NEXT

Chapter 5. Numbers

Chapter Topics

In this chapter, we will focus on Python's numeric types. We will cover each type in detail, then present the various operators and built-in functions that can be used with numbers. We conclude this chapter by introducing some of the standard library modules that deal with numbers.

5.1. Introduction to Numbers

Numbers provide literal or scalar storage and direct access. A number is also an immutable type, meaning that changing or updating its value results in a newly allocated object. This activity is, of course, transparent to both the programmer and the user, so it should not change the way the application is developed.

Python has several numeric types: "plain" integers, long integers, Boolean, double-precision floating point real numbers, decimal floating point numbers, and complex numbers.

How to Create and Assign Numbers (Number Objects)

Creating numbers is as simple as assigning a value to a variable:

```
anInt = 1
aLong = -9999999999999999L
aFloat = 3.14159265358979323846264338327795
aComplex = 1.23+4.56J
```

How to Update Numbers

You can "update" an existing number by (re)assigning a variable to another number. The new value can be related to its previous value or to a completely different number altogether. We put quotes around *update* because you are not really changing the value of the original variable. Because numbers are immutable, you are just making a new number and reassigning the reference. Do not be fooled by what you were taught about how variables contain values that allow you to update them. Python's object model is more specific than that.

When we learned programming, we were taught that variables act like boxes that hold values. In Python, variables act like pointers that point to boxes. For immutable types, you do not change the contents of the box, you just point your pointer at a new box. Every time you assign another number to a variable, you are creating a new object and assigning it. (This is true for all immutable types, not just numbers.)

```
anInt += 1
aFloat = 2.718281828
```

How to Remove Numbers

Under normal circumstances, you do not really "remove" a number; you just stop using it! If you really want to delete a reference to a number object, just use the **`del`** statement (introduced in Section 3.5.6). You can no longer use the variable name, once removed, unless you assign it to a new object; otherwise, you will cause a `NameError` exception to occur.

```
del anInt
del aLong, aFloat, aComplex
```

Okay, now that you have a good idea of how to create and update numbers, let us take a look at Python's four numeric types.

PREV NEXT

5.2. Integers

Python has several types of integers. There is the Boolean type with two possible values. There are the regular or plain integers: generic vanilla integers recognized on most systems today. Python also has a long integer size; however, these far exceed the size provided by C `longs` . We will take a look at these types of integers, followed by a description of operators and built-in functions applicable only to Python integer types.

5.2.1. Boolean

The Boolean type was introduced in Python 2.3. Objects of this type have two possible values, Boolean `TRue` and `False`. We will explore Boolean objects toward the end of this chapter in Section 5.7.1.

5.2.2. Standard (Regular or Plain) Integers

Python's "plain" integers are the universal numeric type. Most machines (32-bit) running Python will provide a range of -2^{31} to $2^{31}-1$, that is -2, 147,483,648 to 2,147,483,647. If Python is compiled on a 64-bit system with a 64-bit compiler, then the integers for that system will be 64-bit. Here are some examples of Python integers:

```
0101    84      -237    0x80    017     -680    -0X92
```

Python integers are implemented as (signed) `long`s in C. Integers are normally represented in base 10 decimal format, but they can also be specified in base 8 or base 16 representation. Octal values have a "0" prefix, and hexadecimal values have either "0x" or "0X" prefixes.

5.2.3. Long Integers

The first thing we need to say about Python long integers (or `long`s for short) is *not* to get them confused with longs in C or other compiled languagesthese values are typically restricted to 32- or 64-bit sizes, whereas Python longs are limited only by the amount of (virtual) memory in your machine. In other words, they can be very L-O-N-G longs.

Longs are a superset of integers and are useful when your application requires integers that exceed the range of plain integers, meaning less than -2^{31} or greater than $2^{31}-1$. Use of longs is denoted by the letter "`L`", uppercase (`L`) or lowercase (`l`), appended to the integer's numeric value. Values can be expressed in decimal, octal, or hexadecimal. The following are examples of longs:

```
16384L     -0x4E8L 017L  -2147483648l   052144364L

299792458l 0xDECADEDEADBEEFBADFEEDDEAL    -5432101234L
```

Core Style: Use uppercase "L" with long integers

Although Python supports a case-insensitive "L" to denote longs, we recommend that you use only the uppercase "L" to avoid confusion with the number one (1). Python will display only longs with a capital "L ." As integers and longs are slowly being unified, you will only see the "L" with evaluatable string representations (`repr()`) of longs. Printable string representations (`str()`) will not have the "L ."

```
>>> aLong = 999999999l
>>> aLong
999999999L
>>> print aLong
999999999
```

5.2.4. Unification of Integers and Long Integers

Both integer types are in the process of being unified into a single integer type. Prior to Python 2.2, plain integer operations resulted in overflow (i.e., greater than the 2^{32} range of numbers described above), but in 2.2 or after, there are no longer such errors.

Python 2.1

```
>>> 9999 ** 8
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
OverflowError: integer exponentiation
```

Python 2.2

```
>>> 9999 ** 8
99920027994400699944002799920001L
```

Removing the error was the first phase. The next step involved bit-shifting; it used to be possible to left-shift bits out of the picture (resulting in 0):

```
>>> 2 << 32
0
```

In 2.3 such an operation gives a warning, but in 2.4 the warning is gone, and the operation results in a

real (long) value:

Python 2.3

```
>>> 2 << 32
__main__:1: FutureWarning: x<<y losing bits or changing
sign will return a long in Python 2.4
and up
0
```

Python 2.4

```
>>> 2 << 32
8589934592L
```

Sooner or later (probably later), there will no longer be a long type (at least not at the user level). Things will all happen quietly under the covers. Of course, those with C access will be able to enjoy both types as before, meaning, however, that your C code will still need to be able to distinguish between the different Python integer types. You can read more about the unification of integers and longs in PEP 237.

PREV NEXT

5.3. Double Precision Floating Point Numbers

Floats in Python are implemented as C `double`s, double precision floating point real numbers, values that can be represented in straightforward decimal or scientific notations. These 8-byte (64-bit) values conform to the IEEE 754 definition (52M/11E/1S) where 52 bits are allocated to the mantissa, 11 bits to the exponent (this gives you about $\pm 10^{308.25}$ in range), and the final bit to the sign. That all sounds fine and dandy; however, the actual degree of precision you will receive (along with the range and overflow handling) depends completely on the architecture of the machine as well as the implementation of the compiler that built your Python interpreter.

Floating point values are denoted by a decimal point (.) in the appropriate place and an optional "e" suffix representing scientific notation. We can use either lowercase (`e`) or uppercase (`E`). Positive (`+`) or negative (`-`) signs between the "e" and the exponent indicate the sign of the exponent. Absence of such a sign indicates a positive exponent. Here are some floating point values:

```
0.0     -777.      1.6       -5.555567119 96e3 * 1.0
4.3e25  9.384e-23  -2.172818 float(12)    1.000000001
3.1416  4.2E-10    -90.      6.022e23     -1.609E-19
```

5.4. Complex Numbers

A long time ago, mathematicians were absorbed by the following equation:

$$x^2 = -1$$

The reason for this is that any real number (positive or negative) multiplied by itself results in a positive number. How can you multiply any number with itself to get a negative number? No such real number exists. So in the eighteenth century, mathematicians invented something called an *imaginary number i* (or *j*, depending on what math book you are reading) such that:

$$j = \sqrt{-1}$$

Basically a new branch of mathematics was created around this special number (or concept), and now imaginary numbers are used in numerical and mathematical applications. Combining a real number with an imaginary number forms a single entity known as a *complex number*. A complex number is any ordered pair of floating point real numbers (`x`, `y`) denoted by `x + yj` where `x` is the real part and `y` is the imaginary part of a complex number.

It turns out that complex numbers are used a lot in everyday math, engineering, electronics, etc. Because it became clear that many researchers were reinventing this wheel quite often, complex numbers became a real Python data type long ago in version 1.4.

Here are some facts about Python's support of complex numbers:

- Imaginary numbers by themselves are not supported in Python (they are paired with a real part of 0.0 to make a complex number)
- Complex numbers are made up of real and imaginary parts
- Syntax for a complex number: *`real+imagj`*
- Both real and imaginary components are floating point values
- Imaginary part is suffixed with letter "J" lowercase (`j`) or uppercase (`J`)

The following are examples of complex numbers:

```
64.375+1j  4.23-8.5j   0.23-8.55j  1.23e-045+6.7e+089j
6.23+1.5j  -1.23-875J 0+1j  9.80665-8.31441J   -.0224+0j
```

5.4.1. Complex Number Built-in Attributes

Complex numbers are one example of objects with data attributes (Section 4.1.1). The data attributes are the real and imaginary components of the complex number object they belong to. Complex numbers also have a method attribute that can be invoked, returning the complex conjugate of the object.

```
>>> aComplex = -8.333-1.47j
>>> aComplex
(-8.333-1.47j)
```

```
>>> aComplex.real
-8.333
>>> aComplex.imag
-1.47
>>> aComplex.conjugate()
(-8.333+1.47j)
```

Table 5.1 describes the attributes of complex numbers.

Table 5.1. Complex Number Attributes

Attribute	*Description*
num.real	Real component of complex number *num*
num.imag	Imaginary component of complex number *num*
num.conjugate()	Returns complex conjugate of *num*

5.5. Operators

Numeric types support a wide variety of operators, ranging from the standard type of operators to operators created specifically for numbers, and even some that apply to integer types only.

5.5.1. Mixed-Mode Operations

It may be hard to remember, but when you added a pair of numbers in the past, what was important was that you got your numbers correct. Addition using the plus (+) sign was always the same. In programming languages, this may not be as straightforward because there are different types of numbers.

When you add a pair of integers, the + represents integer addition, and when you add a pair of floating point numbers, the + represents double-precision floating point addition, and so on. Our little description extends even to non-numeric types in Python. For example, the + operator for strings represents concatenation, not addition, but it uses the same operator! The point is that for each data type that supports the + operator, there are different pieces of functionality to "make it all work," embodying the concept of *overloading*.

Now, we cannot add a number and a string, but Python does support mixed mode operations strictly between numeric types. When adding an integer and a float, a choice has to be made as to whether integer or floating point addition is used. There is no hybrid operation. Python solves this problem using something called *numeric coercion*. This is the process whereby one of the operands is converted to the same type as the other before the operation. Python performs this coercion by following some basic rules.

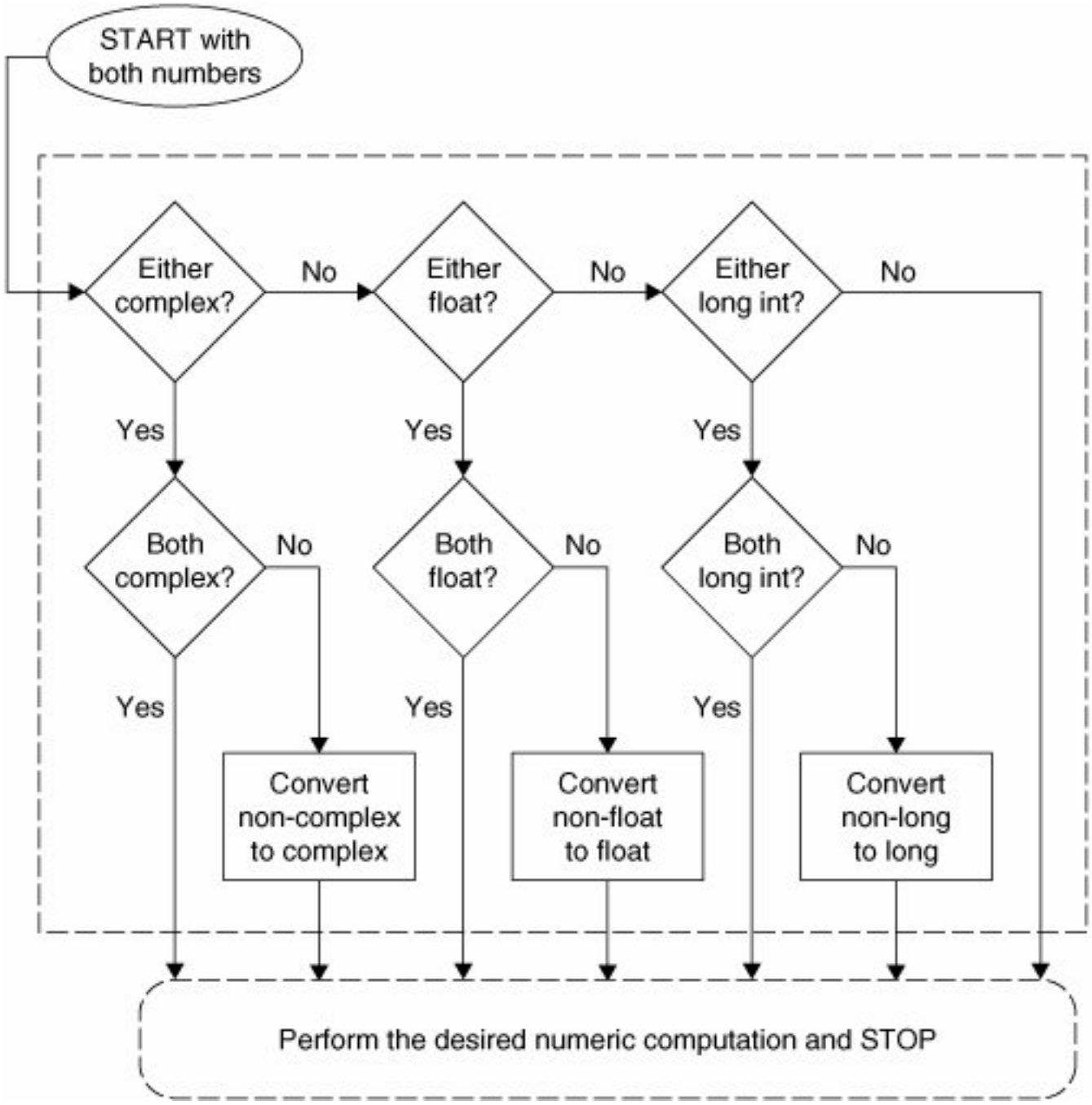
To begin with, if both numbers are the same type, no conversion is necessary. When both types are different, a search takes place to see whether one number can be converted to the other's type. If so, the operation occurs and both numbers are returned, one having been converted. There are rules that must be followed since certain conversions are impossible, such as turning a float into an integer, or converting a complex number to any non-complex number type.

Coercions that are possible, however, include turning an integer into a float (just add "`.0`") or converting any non-complex type to a complex number (just add a zero imaginary component, e.g., "`0j`"). The rules of coercion follow from these two examples: integers move toward float, and all move toward complex. The Python Language Reference Guide describes the `coerce()` operation in the following manner.

- If either argument is a complex number, the other is converted to complex;
- Otherwise, if either argument is a floating point number, the other is converted to floating point;
- Otherwise, if either argument is a long, the other is converted to long;
- Otherwise, both must be plain integers and no conversion is necessary (in the upcoming diagram, this describes the rightmost arrow).

The flowchart shown in Figure 5-1 illustrates these coercion rules.

Figure 5-1. Numeric coercion



Automatic numeric coercion makes life easier for the programmer because he or she does not have to worry about adding coercion code to his or her application. If explicit coercion is desired, Python does provide the `coerce()` built-in function (described later in Section 5.6.2).

The following is an example showing you Python's automatic coercion. In order to add the numbers (one integer, one float), both need to be converted to the same type. Since float is the superset, the integer is coerced to a float before the operation happens, leaving the result as a float:

```
>>> 1 + 4.5
5.5
```

5.5.2. Standard Type Operators

The standard type operators discussed in Chapter 4 all work as advertised for numeric types. Mixed-mode operations, described above, are those which involve two numbers of different types. The values

are internally converted to the same type before the operation is applied.

Here are some examples of the standard type operators in action with numbers:

```
>>> 5.2 == 5.2
True
>>> -719 >= 833
False
>>> 5+4e >= 2-3e
True
>>> 2 < 5 < 9      # same as ( 2 < 5 )and ( 5 < 9 )
True
>>> 77 > 66 == 66    # same as ( 77 > 66 )and ( 66 == 66 )
True
>>> 0. < -90.4 < 55.3e2 != 3 < 181
False
>>> (-1 < 1) or (1 < -1)
True
```

5.5.3. Numeric Type (Arithmetic) Operators

Python supports unary operators for no change and negation, `+` and `-`, respectively; and binary arithmetic operators `+`, `-`, `*`, `/`, `%`, and `**`, for addition, subtraction, multiplication, division, modulo, and exponentiation, respectively. In addition, there is a new division operator, `//`, as of Python 2.2.

Division

Those of you coming from the C world are intimately familiar with *classic division*that is, for integer operands, *floor division* is performed, while for floating point numbers, real or *true division* is the operation. However, for those who are learning programming for the first time, or for those who rely on accurate calculations, code must be tweaked in a way to obtain the desired results. This includes casting or converting all values to floats before performing the division.

The decision has been made to change the division operator in some future version of Python from classic to true division and add another operator to perform floor division. We now summarize the various division types and show you what Python currently does, and what it will do in the future.

Classic Division

When presented with integer operands, classic division truncates the fraction, returning an integer (floor division). Given a pair of floating-point operands, it returns the actual floating-point quotient (true division). This functionality is standard among many programming languages, including Python. Example:

```
>>> 1 / 2              # perform integer result (floor)
0
>>> 1.0 / 2.0          # returns actual quotient
0.5
```

True Division

This is where division always returns the actual quotient, regardless of the type of the operands. In a future version of Python, this will be the algorithm of the division operator. For now, to take advantage of true division, one must give the `from__future__import division` directive. Once that happens, the division operator (`/`) performs only true division:

```
>>> from __future__ import division
>>>
>>> 1 / 2              # returns real quotient
0.5
>>> 1.0 / 2.0          # returns real quotient
0.5
```

Floor Division

A new division operator (`//`) has been created that carries out floor division: it always truncates the fraction and rounds it to the next smallest whole number toward the left on the number line, regardless of the operands' numeric types. This operator works starting in 2.2 and does not require the `__future__` directive above.

```
>>> 1 // 2     # floors result, returns integer
0
>>> 1.0 // 2.0 # floors result, returns float
0.0
>>> -1 // 2    # move left on number line
-1
```

There were strong arguments for as well as against this change, with the former from those who want or need true division versus those who either do not want to change their code or feel that altering the division operation from classic division is wrong.

This change was made because of the feeling that perhaps Python's division operator has been flawed from the beginning, especially because Python is a strong choice as a first programming language for people who aren't used to floor division. One of van Rossum's use cases is featured in his "What's New in Python 2.2" talk:

```
def velocity(distance, totalTime):
    rate = distance / totalTime
```

As you can tell, this function may or may not work correctly and is solely dependent on at least one argument being a floating point value. As mentioned above, the only way to ensure the correct value is to cast both to floats, i.e., `rate = float(distance) / float(totalTime)`. With the upcoming change to true division, code like the above can be left as is, and those who truly desire floor division can use the new double-slash (`//`) operator.

Yes, code breakage is a concern, and the Python team has created a set of scripts that will help you convert your code to using the new style of division. Also, for those who feel strongly either way and

only want to run Python with a specific type of division, check out the `-Qdivision_style` option to the interpreter. An option of `-Qnew` will always perform true division while `-Qold` (currently the default) runs classic division. You can also help your users transition to new division by using `-Qwarn` or `-Qwarnall`.

More information about this big change can be found in PEP 238. You can also dig through the 2001 `comp.lang.python` archives for the heated debates if you are interested in the drama. Table 5.2 summarizes the division operators in the various releases of Python and the differences in operation when you import new division functionality.

Table 5.2. Division Operator Functionality

Operator	*2.1.x and Older*	*2.2 and Newer (No Import)*	*2.2 and Newer (Import of `division`)*
`/`	classic	classic	true
`//`	n/a	floor	floor

Modulus

Integer modulo is straightforward integer division remainder, while for float, it is the difference of the dividend and the product of the divisor and the quotient of the quantity dividend divided by the divisor rounded down to the closest integer, i.e.,`x - (math.floor(x/y) * y)`, or

$$x - \left\lfloor \frac{x}{y} \right\rfloor \times y$$

For complex number modulo, take only the real component of the division result, i.e., `x - (math.floor((x/y).real) * y)`.

Exponentiation

The exponentiation operator has a peculiar precedence rule in its relationship with the unary operators: it binds more tightly than unary operators to its left, but less tightly than unary operators to its right. Due to this characteristic, you will find the `**` operator twice in the numeric operator charts in this text. Here are some examples:

```
>>> 3 ** 2
9
>>> -3 ** 2       # ** binds tighter than - to its left
-9
>>> (-3) ** 2     # group to cause - to bind first
9
>>> 4.0 ** -1.0   # ** binds looser than - to its right
0.25
```

In the second case, it performs 3 to the power of 2 (3-squared) before it applies the unary negation. We need to use the parentheses around the "-3" to prevent this from happening. In the final example, we

see that the unary operator binds more tightly because the operation is 1 over quantity 4 to the first power ¼1 or ¼. Note that 1 / 4 as an integer operation results in an integer 0, so integers are not allowed to be raised to a negative power (it is a floating point operation anyway), as we will show here:

```
>>> 4 ** -1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: integer to the negative power
```

Summary

Table 5.3 summarizes all arithmetic operators, in shaded hierarchical order from highest-to-lowest priority. All the operators listed here rank higher in priority than the bitwise operators for integers found in Section 5.5.4.

Table 5.3. Numeric Type Arithmetic Operators

Arithmetic Operator	*Function*
*expr1 ** expr2*	*expr1* raised to the power of *expr2*[a]
+expr	(unary) *expr* sign unchanged
-expr	(unary) negation of *expr*
*expr1 ** expr2*	*expr1* raised to the power of *expr2*[a]
*expr1 * expr2*	*expr1* times expr2
expr1 / expr2	*expr1* divided by *expr2* (classic or true division)
expr1 // expr2	*expr1* divided by *expr2* (floor division [only])
expr1 % expr2	*expr1* modulo *expr2*
expr1 + expr2	*expr1* plus *expr2*
expr1 - expr2	expr1 minus expr2

[a] ** binds tighter than unary operators to its left and looser than unary operators to its right.

Here are a few more examples of Python's numeric operators:

```
>>> -442 - 77
-519
>>>
>>> 4 ** 3
64
>>>
>>> 4.2 ** 3.2
```

```
98.7183139527
>>> 8 / 3
2
>>> 8.0 / 3.0
2.66666666667
>>> 8 % 3
2
>>> (60. - 32.) * ( 5. / 9. )
15.5555555556
>>> 14 * 0x04
56
>>> 0170 / 4

30
>>> 0x80 + 0777
639
>>> 45L * 22L
990L
>>> 16399L + 0xA94E8L
709879L
>>> -2147483648L - 52147483648L
-54294967296L
>>> 64.375+1j + 4.23-8.5j
(68.605-7.5j)
>>> 0+1j ** 2          # same as 0+(lj**2)
(-1+0j)
>>> 1+1j ** 2          # same as 1+(lj**2)
0j
>>> (1+1j) ** 2
2j
```

Note how the exponentiation operator is still higher in priority than the binding addition operator that delimits the real and imaginary components of a complex number. Regarding the last example above, we grouped the components of the complex number together to obtain the desired result.

5.5.4. *Bit Operators (Integer-Only)

Python integers may be manipulated bitwise and the standard bit operations are supported: inversion, bitwise AND, OR, and exclusive OR (aka XOR), and left and right shifting. Here are some facts regarding the bit operators:

- Negative numbers are treated as their 2's complement value.
- Left and right shifts of N bits are equivalent to multiplication and division by (2 ** N) without overflow checking.
- For longs, the bit operators use a "modified" form of 2's complement, acting as if the sign bit were extended infinitely to the left.

The bit inversion operator (~) has the same precedence as the arithmetic unary operators, the highest of all bit operators. The bit shift operators (<< and >>) come next, having a precedence one level below that of the standard plus and minus operators, and finally we have the bitwise AND, XOR, and OR operators (&, ^, |), respectively. All of the bitwise operators are presented in the order of descending priority in Table 5.4.

Table 5.4. Integer Type Bitwise Operators

Bitwise Operator	*Function*
~num	(unary) invert the bits of num, yielding -(*num* + 1)
num1 << num2	*num1* left shifted by *num2* bits
num1 >> num2	*num1* right shifted by *num2* bits
num1 & num2	*num1* bitwise AND with *num2*
num1 ^ num2	*num1* bitwise XOR (exclusive OR) with *num2*
num1 | num2	*num1* bitwise OR with *num2*

Here we present some examples using the bit operators using 30 (011110), 45 (101101), and 60 (111100):

```
>>> 30 & 45
12
>>> 30 | 45
63
>>> 45 & 60
44
>>> 45 | 60
61
>>> ~30
-31
>>> ~45
-46
>>> 45 << 1
90
>>> 60 >> 2
15
>>> 30 ^ 45
51
```

PREV NEXT

5.6. Built-in and Factory Functions

5.6.1. Standard Type Functions

In the last chapter, we introduced the `cmp()`, `str()`, and `type()` built-in functions that apply for all standard types. For numbers, these functions will compare two numbers, convert numbers into strings, and tell you a number's type, respectively. Here are some examples of using these functions:

```
>>> cmp(-6, 2)
-1
>>> cmp(-4.333333, -2.718281828)
-1
>>> cmp(0xFF, 255)
0
>>> str(0xFF)
'255'
>>> str(55.3e2)
'5530.0'
>>> type(0xFF)
<type 'int'>
>>> type(98765432109876543210L)
<type 'long'>
>>> type(2-1j)
<type 'complex'>
```

5.6.2. Numeric Type Functions

Python currently supports different sets of built-in functions for numeric types. Some convert from one numeric type to another while others are more operational, performing some type of calculation on their numeric arguments.

Conversion Factory Functions

The `int()`, `long()`, `float()`, and `complex()` functions are used to convert from any numeric type to another. Starting in Python 1.5, these functions will also take strings and return the numerical value represented by the string. Beginning in 1.6, `int()` and `long()` accepted a base parameter (see below) for proper string conversionsit does not work for numeric type conversion.

A fifth function, `bool()`, was added in Python 2.2. At that time, it was used to normalize Boolean values to their integer equivalents of one and zero for true and false values. The Boolean type was added in Python 2.3, so true and false now had constant values of `TRue` and `False` (instead of one and zero). For more information on the Boolean type, see Section 5.7.1.

In addition, because of the unification of types and classes in Python 2.2, all of these built-in functions were converted into factory functions. Factory functions, introduced in Chapter 4, just means that these objects are now classes, and when you "call" them, you are just creating an instance of that class.

They will still behave in a similar way to the new Python user so it is probably something you do not have to worry about.

The following are some examples of using these functions:

```
>>> int(4.25555)
4
>>> long(42)
42L
>>> float(4)
4.0
>>> complex(4)
(4+0j)
>>>
>>> complex(2.4, -8)
(2.4-8j)
>>>
>>> complex(2.3e-10, 45.3e4)
(2.3e-10+453000j)
```

Table 5.5 summarizes the numeric type factory functions.

Table 5.5. Numeric Type Factory Functions[a]

Class (Factory Function)	*Operation*
bool(obj)[b]	Returns the Boolean value of *obj*, e.g., the value of executing *obj.__nonzero__()*
int*(obj, base=10)*	Returns integer representation of string or number *obj*; similar to `string.atoi()`; optional *base* argument introduced in 1.6
long*(obj, base=10)*	Returns long representation of string or number *obj*; similar to `string.atol()`; optional *base* argument introduced in 1.6
float*(obj)*	Returns floating point representation of string or number *obj*; similar to `string.atof()`
complex*(str)* **or** *complex(real, imag=0.0)*	Returns complex number representation of `str`, or builds one given *real* (and perhaps *imag*inary) component(s)

[a] Prior to Python 2.3, these were all built-in functions.

[b] New in Python 2.2 as built-in function, converted to factory function in 2.3.

Operational

Python has five operational built-in functions for numeric types: `abs()`, `coerce()`, `divmod()`, `pow()`, and `round()`. We will take a look at each and present some usage examples.

`abs()` returns the absolute value of the given argument. If the argument is a complex number, then `math.sqrt(`*`num`*` .real`^{2}` + `*`num`*`.imag`^{2}`) is returned. Here are some examples of using the `abs()` built-in function:

```
>>> abs(-1)
1
>>> abs(10.)
10.0
>>> abs(1.2-2.1j)
2.41867732449
>>> abs(0.23 - 0.78)
0.55
```

The `coerce()` function, although it technically is a numeric type conversion function, does not convert to a specific type and acts more like an operator, hence our placement of it in our operational built-ins section. In Section 5.5.1, we discussed numeric coercion and how Python performs that operation. The `coerce()` function is a way for the programmer to explicitly coerce a pair of numbers rather than letting the interpreter do it. This feature is particularly useful when defining operations for newly created numeric class types. `coerce()` just returns a tuple containing the converted pair of numbers. Here are some examples:

```
>>> coerce(1, 2)
(1, 2)
>>>
>>> coerce(1.3, 134L)
(1.3, 134.0)
>>>
>>> coerce(1, 134L)
(1L, 134L)
>>>
>>> coerce(1j, 134L)
(1j, (134+0j))
>>>
>>> coerce(1.23-41j, 134L)
((1.23-41j), (134+0j))
```

The `divmod()` built-in function combines division and modulus operations into a single function call that returns the pair (quotient, remainder) as a tuple. The values returned are the same as those given for the classic division and modulus operators for integer types. For floats, the quotient returned is `math.floor(`*`num1/num2`*`)` and for complex numbers, the quotient is `math.floor((`*`num1/num2`*`).real)`.

```
>>> divmod(10,3)
(3, 1)
>>> divmod(3,10)
(0, 3)
```

```
>>> divmod(10,2.5)
(4.0, 0.0)
>>> divmod(2.5,10)
(0.0, 2.5)
>>> divmod(2+1j, 0.5-1j)
(0j, (2+1j))
```

Both `pow()` and the double star (`**`) operator perform exponentiation; however, there are differences other than the fact that one is an operator and the other is a built-in function.

The `**` operator did not appear until Python 1.5, and the `pow()` built-in takes an optional third parameter, a modulus argument. If provided, `pow()` will perform the exponentiation first, then return the result modulo the third argument. This feature is used for cryptographic applications and has better performance than `pow(x,y) % z` since the latter performs the calculations in Python rather than in C-like `pow(x, y, z)`.

```
>>> pow(2,5)
32
>>>s
>>> pow(5,2)
25
>>> pow(3.141592,2)
9.86960029446
>>>
>>> pow(1+1j, 3)
(-2+2j)
```

The `round()` built-in function has a syntax of `round(flt,ndig=0)`. It normally rounds a floating point number to the nearest integral number and returns that result (still) as a float. When the optional *ndig* option is given, `round()` will round the argument to the specific number of decimal places.

```
>>> round(3)
3.0
>>> round(3.45)
3.0
>>> round(3.4999999)
3.0
>>> round(3.4999999, 1)
3.5
>>> import math
>>> for eachNum in range(10):
...           print round(math.pi, eachNum)
...
3.0
3.1
3.14
3.142
3.1416
3.14159
3.141593
3.1415927
3.14159265
3.141592654
3.1415926536
```

```
>>> round(-3.5)
-4.0
>>> round(-3.4)
-3.0
>>> round(-3.49)
-3.0
>>> round(-3.49, 1)
-3.5
```

Note that the rounding performed by `round()` moves away from zero on the number line, i.e., `round(.5)` goes to 1 and `round(-.5)` goes to -1. Also, with functions like `int()`, `round()`, and `math.floor()`, all may seem like they are doing the same thing; it is possible to get them all confused. Here is how you can differentiate among these:

- `int()` chops off the decimal point and everything after (aka truncation).
- `floor()` rounds you to the next smaller integer, i.e., the next integer moving in a negative direction (toward the left on the number line).
- `round()` (rounded zero digits) rounds you to the nearest integer period.

Here is the output for four different values, positive and negative, and the results of running these three functions on eight different numbers. (We reconverted the result from `int()` back to a float so that you can visualize the results more clearly when compared to the output of the other two functions.)

```
>>> import math
>>> for eachNum in (.2, .7, 1.2, 1.7, -.2, -.7, -1.2, -1.7):
...     print "int(%.1f)\t%+.1f" % (eachNum, float(int(eachNum)))
...     print "floor(%.1f)\t%+.1f" % (eachNum,
...     math.floor(eachNum))
...     print "round(%.1f)\t%+.1f" % (eachNum, round(eachNum))
...     print '-' * 20
...
int(0.2)     +0.0
floor(0.2)   +0.0
round(0.2)   +0.0
--------------------
int(0.7)     +0.0
floor(0.7)   +0.0
round(0.7)   +1.0
--------------------
int(1.2)     +1.0
floor(1.2)   +1.0
round(1.2)   +1.0
--------------------
int(1.7)     +1.0
floor(1.7)   +1.0
round(1.7)   +2.0
--------------------
int(-0.2)    +0.0
floor(-0.2)  -1.0
round(-0.2)  +0.0
--------------------
int(-0.7)    +0.0
floor(-0.7)  -1.0
round(-0.7)  -1.0
--------------------
int(-1.2)    -1.0
```

```
floor(-1.2)  -2.0
round(-1.2)  -1.0
--------------------
int(-1.7)    -1.0
floor(-1.7)  -2.0
round(-1.7)  -2.0
```

Table 5.6 summarizes the operational functions for numeric types.

Table 5.6. Numeric Type Operational Built-in Functions[a]

Function	*Operation*
`abs(num)`	Returns the absolute value of *num*
`coerce(num1, num2)`	Converts *num1* and *num2* to the same numeric type and returns the converted pair as a tuple
`divmod(num1, num2)`	Division-modulo combination returns (*num1 / num2, num1 % num2*) as a tuple; for floats and complex, the quotient is rounded down (complex uses only real component of quotient)
`pow(num1, num2, mod=1)`	Raises *num1* to *num2* power, quantity modulo *mod* if provided
`round(flt, ndig=0)`	(Floats only) takes a float *flt* and rounds it to *ndig* digits, defaulting to zero if not provided

[a] Except for `round()`, which applies only to floats.

5.6.3. Integer-Only Functions

In addition to the built-in functions for all numeric types, Python supports a few that are specific only to integers (plain and long). These functions fall into two categories, base presentation with `hex()` and `oct()`, and ASCII conversion featuring `chr()` and `ord()`.

Base Representation

As we have seen before, Python integers automatically support octal and hexadecimal representations in addition to the decimal standard. Also, Python has two built-in functions that return string representations of an integer's octal or hexadecimal equivalent. These are the `oct()` and `hex()` built-in functions, respectively. They both take an integer (in any representation) object and return a string with the corresponding value. The following are some examples of their usage:

```
>>> hex(255)
'0xff'
>>> hex(23094823l)
'0x1606627L'
>>> hex(65535*2)
'0x1fffe'
```

```
>>>
>>> oct(255)
'0377'
>>> oct(23094823l)
'0130063047L'
>>> oct(65535*2)
'0377776'
```

ASCII Conversion

Python also provides functions to go back and forth between ASCII (American Standard Code for Information Interchange) characters and their ordinal integer values. Each character is mapped to a unique number in a table numbered from 0 to 255. This number does not change for all computers using the ASCII table, providing consistency and expected program behavior across different systems. `chr()` takes a single-byte integer value and returns a one-character string with the equivalent ASCII character. `ord()` does the opposite, taking a single ASCII character in the form of a string of length one and returns the corresponding ASCII value as an integer:

```
>>> ord('a')
97
>>> ord('A')
65
>>> ord('0')
48

>>> chr(97)
'a'
>>> chr(65L)
'A'
>>> chr(48)
'0'
```

Table 5.7 shows all built-in functions for integer types.

Table 5.7. Integer Type Built-in Functions

Function	*Operation*
`hex(num)`	Converts *num* to hexadecimal and returns as string
`oct(num)`	Converts *num* to octal and returns as string
`chr(num)`	Takes ASCII value *num* and returns ASCII character as string; 0 <= *num* <= 255 only
`ord(chr)`	Takes ASCII or Unicode *chr* (string of length 1) and returns corresponding ordinal ASCII value or Unicode code point, respectively
`unichr(num)`	Takes a Unicode code point value num and returns its Unicode character as a Unicode string; valid range depends on whether your Python was built as UCS-2 or UCS-4

5.7. Other Numeric Types

5.7.1. Boolean "Numbers"

2.3

Boolean types were added to Python starting in version 2.3. Although Boolean values are spelled "True" and "False," they are actually an integer subclass and will behave like integer values one and zero, respectively, if used in a numeric context. Here are some of the major concepts surrounding Boolean types:

- They have a constant value of either `true` or `False`.
- Booleans are subclassed from integers but cannot themselves be further derived.
- Objects that do not have a `__nonzero__()` method default to `true`.
- Recall that Python objects typically have a Boolean `False` value for any numeric zero or empty set.
- Also, if used in an arithmetic context, Boolean values `TRue` and `False` will take on their numeric equivalents of 1 and 0, respectively.
- Most of the standard library and built-in Boolean functions that previously returned integers will now return Booleans.
- Neither `TRue` nor `False` are keywords yet but will be in a future version.

All Python objects have an inherent `true` or `False` value. To see what they are for the built-in types, review the Core Note sidebar in Section 4.3.2. Here are some examples using Boolean values:

```
# intro
>>> bool(1)
True
>>> bool(True)
True
>>> bool(0)
False
>>> bool('1')
True
>>> bool('0')
True
>>> bool([])
False
>>> bool ( (1,) )
True

# using Booleans numerically
>>> foo = 42
>>> bar = foo < 100
>>> bar
True
>>> print bar + 100
101
>>> print '%s' % bar
True
```

```
>>> print '%d' % bar
1

# no __nonzero__()
>>> class C: pass
>>> c = C()
>>>
>>> bool(c)
True
>>> bool(C)
True

# __nonzero__() overridden to return False
>>> class C:
...     def __nonzero__(self):
...         return False
...
>>> c = C()
>>> bool(c)
False
>>> bool(C)
True

# OH NO!! (do not attempt)
>>> True, False = False, True
>>> bool(True)
False
>>> bool(False)
True
```

You can read more about Booleans in the Python documentation and PEP 285.

5.7.2. Decimal Floating Point Numbers

Decimal floating point numbers became a feature of Python in version 2.4 (see PEP 327), mainly because statements like the following drive many (scientific and financial application) programmers insane (or at least enrage them):

```
>>> 0.1
0.1000000000000001
```

2.4

Why is this? The reason is that most implementations of doubles in C are done as a 64-bit IEEE 754 number where 52 bits are allocated for the mantissa. So floating point values can only be specified to 52 bits of precision, and in situations where you have a(n endlessly) repeating fraction, expansions of such values in binary format are snipped after 52 bits, resulting in rounding errors like the above. The value .1 is represented by 0.11001100110011 ... * 2^{-3} because its closest binary approximation is .0001100110011 ..., or 1/16 + 1/32 + 1/256 + ...

As you can see, the fractions will continue to repeat and lead to the rounding error when the repetition

cannot "be continued." If we were to do the same thing using a decimal number, it looks much "better" to the human eye because they have exact and arbitrary precision. Note in the below that you cannot mix and match decimals and floating point numbers. You can create decimals from strings, integers, or other decimals. You must also import the `decimal` module to use the `Decimal` number class.

```
>>> from decimal import Decimal
>>> dec = Decimal(.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/usr/local/lib/python2.4/decimal.py", line 523, in __new__
    raise TypeError("Cannot convert float to Decimal.  " +
TypeError: Cannot convert float to Decimal.  First convert the float to
a string
>>> dec = Decimal('.1')
>>> dec
Decimal("0.1")
>>> print dec
0.1
>>> dec + 1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/usr/local/lib/python2.4/decimal.py", line 906, in __add__
    other = _convert_other(other)
  File "/usr/local/lib/python2.4/decimal.py", line 2863, in
_convert_other
    raise TypeError, "You can interact Decimal only with int, long or
Decimal data types."
TypeError: You can interact Decimal only with int, long or Decimal data
types.
>>>
>>> dec + Decimal('1.0')
Decimal("1.1")
>>> print dec + Decimal('1.0')
1.1
```

You can read more about decimal numbers in the PEP as well as the Python documentation, but suffice it to say that they share pretty much the same numeric operators as the standard Python number types. Since it is a specialized numeric type, we will not include decimals in the remainder of this chapter.

5.8. Related Modules

There are a number of modules in the Python standard library that add on to the functionality of the operators and built-in functions for numeric types. Table 5.8 lists the key modules for use with numeric types. Refer to the literature or online documentation for more information on these modules.

Table 5.8. Numeric Type Related Modules

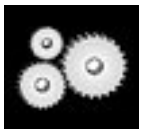
Module	*Contents*
`decimal`	Decimal floating point class Decimal
`array`	Efficient arrays of numeric values (characters, ints, floats, etc.)
`math/cmath`	Standard C library mathematical functions; most functions available in `math` are implemented for complex numbers in the `cmath` module
`operator`	Numeric operators available as function calls, i.e., `operator.sub(m, n)` is equivalent to the difference `(m - n)` for numbers `m` and `n`
`random`	Various pseudo-random number generators (obsoletes `rand` and `wHRandom`)

For advanced numerical and scientific mathematics applications, there are well-known third-party packages Numeric (NumPy) and SciPy, which may be of interest to you. More information on those two packages can be found at:

http://numeric.scipy.org/

http://scipy.org/

Core Module: `random`



The `random` module is the general-purpose place to go if you are looking for random numbers. This module comes with various pseudo-random number generators and comes seeded with the current timestamp so it is ready to go as soon as it has loaded. Here are some of the most commonly used functions in the `random` module:

`randint()`	*Takes two integer values and returns a random integer between those values inclusive*
`randrange()`	*Takes the same input as `range()` and returns a random integer that falls within that range*

`uniform()`	*Does almost the same thing as `randint()`, but returns a float and is inclusive only of the smaller number (exclusive of the larger number)*
`random()`	*Works just like `uniform()` except that the smaller number is fixed at 0.0, and the larger number is fixed at 1.0*
`choice()`	*Given a sequence (see Chapter 6), randomly selects and returns a sequence item*

We have now come to the conclusion of our tour of all of Python's numeric types. A summary of operators and built-in functions for numeric types is given in Table 5.9.

Table 5.9. Operators and Built-in Functions for All Numeric Types

Operator/Built-in	*Description*	*Int*	*Long*	*Float*	*Complex*	*Result*[a]
`abs()`	Absolute value	•	•	•	•	*number*[a]
`chr()`	Character	•	•			`str`
`coerce()`	Numeric coercion	•	•	•	•	`tuple`
`complex()`	Complex factory function	•	•	•	•	`complex`
`divmod()`	Division/modulo	•	•	•	•	`tuple`
`float()`	Float factory function	•	•	•	•	`float`
`hex()`	Hexadecimal string	•	•			`str`
`int()`	Int factory function	•	•	•	•	`int`
`long()`	Long factory function	•	•	•	•	`long`
`oct()`	Octal string	•	•			`str`
`ord()`	Ordinal			`(str)`		`int`
`pow()`	Exponentiation	•	•	•	•	*number*
`round()`	Float rounding			•		`float`
`**`[b]	Exponentiation	•	•	•	•	*number*
`+`[c]	No change	•	•	•	•	*number*

- [c]	Negation	•	•	•	•	*number*
~ [c]	Bit inversion	•	•			`int/long`
** [b]	Exponentiation	•	•	•	•	*number*
*	Multiplication	•	•	•	•	*number*
/	Classic or true division	•	•	•	•	*number*
//	Floor division	•	•	•	•	*number*
%	Modulo/remainder	•	•	•	•	*number*
+	Addition	•	•	•	•	*number*
-	Subtraction	•	•	•	•	*number*
<<	Bit left shift	•	•			`int/long`
>>	Bit right shift	•	•			`int/long`
&	Bitwise AND	•	•			`int/long`
^	Bitwise XOR	•	•			`int/long`
|	Bitwise OR	•	•			`int/long`

[a] A result of "number" indicates any of the four numeric types, perhaps the same as the operands.

[b] ** has a unique relationship with unary operators; see Section 5.5.3 and Table 5.2.

[c] Unary operator.

5.9. Exercises

The exercises in this chapter may first be implemented as applications. Once full functionality and correctness have been verified, we recommend that the reader convert his or her code to functions that can be used in future exercises. On a related note, one style suggestion is not to use `print` statements in functions that return a calculation. The caller can perform any output desired with the return value. This keeps the code adaptable and reusable.

5-1. *Integers.* Name the differences between Python's regular and long integers.

5-2. *Operators.*

a.

Create a function to calculate and return the product of two numbers.

b.

The code which calls this function should display the result.

5-3. *Standard Type Operators.* Take test score input from the user and output letter grades according to the following grade scale/curve:

A.

90-100

B.

80-89

C.

70-79

D.

60-69

E.

<60

5-4. *Modulus*. Determine whether a given year is a leap year, using the following formula: a leap year is one that is divisible by four, but not by one hundred, unless it is also divisible by four hundred. For example, 1992, 1996, and 2000 are leap years, but 1967 and 1900 are not. The next leap year falling on a century is 2400.

5-5. *Modulus*. Calculate the number of basic American coins given a value less than 1 dollar. A penny is worth 1 cent, a nickel is worth 5 cents, a dime is worth 10 cents, and a quarter is worth 25 cents. It takes 100 cents to make 1 dollar. So given an amount less than 1 dollar (if using floats, convert to integers for this exercise), calculate the number of each type of coin necessary to achieve the amount, maximizing the number of larger denomination coins. For example, given $0.76, or 76 cents, the correct output would be "3 quarters and 1 penny." Output such as "76 pennies" and "2 quarters, 2 dimes, 1 nickel, and 1 penny" are not acceptable.

5-6. *Arithmetic*. Create a calculator application. Write code that will take two numbers and an operator in the format: N1 OP N2, where N1 and N2 are floating point or integer values, and OP is one of the following: `+`, `-`, `*`, `/`, `%`, `**`, representing addition, subtraction, multiplication, division, modulus/remainder, and exponentiation, respectively, and displays the result of carrying out that operation on the input operands. Hint: You may use the string `split()` method, but you cannot use the `eval()` built-in function.

5-7. *Sales Tax*. Take a monetary amount (i.e., floating point dollar amount [or whatever currency you use]), and determine a new amount figuring all the sales taxes you must pay where you live.

5-8. *Geometry*. Calculate the area and volume of:

a.

squares and cubes

b.

circles and spheres

5-9. *Style*. Answer the following numeric format questions:

a.

Why does 17 + 32 give you 49, but 017 + 32 give you 47 and 017 + 032 give you 41, as indicated in the examples below?

```
>>> 17 + 32
49
>>> 017+ 32
47
>>> 017 + 032
41
```

b.

Why do we get 134L and not 1342 in the example below?

```
>>> 56l + 78l
134L
```

5-10. *Conversion*. Create a pair of functions to convert Fahrenheit to Celsius temperature values. `C = (F - 32) * (5 / 9 )` should help you get started. We recommend you try true division with this exercise, otherwise take whatever steps are necessary to ensure accurate results.

5-11. *Modulus*.

a.

Using loops and numeric operators, output all even numbers from 0 to 20.

b.

Same as part (a), but output all odd numbers up to 20.

c.

From parts (a) and (b), what is an easy way to tell the difference between even and odd numbers?

d.

Using part (c), write some code to determine if one number divides another. In your solution, ask the user for both numbers and have your function answer "yes" or "no" as to whether one number divides another by returning `TRue` or `False`, respectively.

5-12. *Limits*. Determine the largest and smallest ints, floats, and complex numbers that your system can handle.

5-13. *Conversion*. Write a function that will take a time period measured in hours and minutes and return the total time in minutes only.

5-14. *Bank Account Interest*. Create a function to take an interest percentage rate for a bank account, say, a Certificate of Deposit (CD). Calculate and return the Annual Percentage Yield (APY) if the account balance was compounded daily.

5-15. *GCD and LCM*. Determine the greatest common divisor and least common multiple of a pair of integers.

5-16. *Home Finance*. Take an opening balance and a monthly payment. Using a loop, determine remaining balances for succeeding months, including the final payment. "Payment 0" should just be the opening balance and schedule monthly payment amount. The output should be in a schedule format similar to the following (the numbers used in this example are for illustrative purposes only):

```
Enter opening balance:100.00
Enter monthly payment: 16.13

         Amount    Remaining
Pymt#     Paid      Balance
-----    ------    ---------
  0      $ 0.00     $100.00
  1      $16.13     $ 83.87
  2      $16.13     $ 67.74
  3      $16.13     $ 51.61
  4      $16.13     $ 35.48
  5      $16.13     $ 19.35
  6      $16.13     $  3.22
  7      $ 3.22     $  0.00
```

5-17. **Random Numbers*. Read up on the `random` module and do the following problem: Generate a list of a random number (`1 < N <= 100`) of random numbers (`0 <= n <= 2^31-1`). Then randomly select a set of these numbers (`1 <= N <= 100`), sort them, and display this subset.

PREV NEXT

Chapter 6. Sequences: Strings, Lists, and Tuples

Chapter Topics

The next family of Python types we will be exploring are those whose items are ordered sequentially and accessible via index offsets into its set of elements. This group, known as *sequences*, includes the following types: strings (regular and unicode), lists, and tuples.

We call these sequences because they are made up of sequences of "items" making up the entire data structure. For example, a string consists of a sequence of characters (even though Python does not have an explicit character type), so the first character of a string `"Hello"` is `'H'`, the second character is `'e'`, and so on. Likewise, lists and tuples are sequences of various Python objects.

We will first introduce all operators and built-in functions (BIFs) that apply to all sequences, then cover each type individually. For each sequence type, we will detail the following:

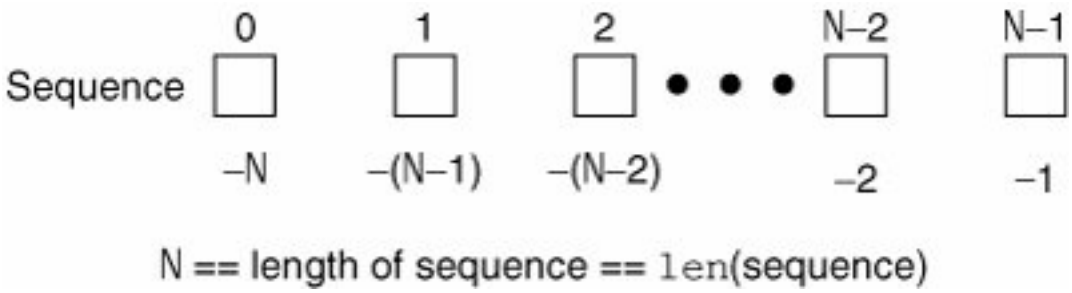
- Introduction
- Operators
- Built-in functions
- Built-in methods (*if applicable*)
- Special features (*if applicable*)
- Related modules (*if applicable*)

We will conclude this chapter with a reference chart that summarizes all of the operators and functions applicable to all sequences. Let us begin by taking a high-level overview.

6.1. Sequences

Sequence types all share the same access model: ordered set with sequentially indexed offsets to get to each element. Multiple elements may be selected by using the slice operators, which we will explore in this chapter. The numbering scheme used starts from zero (0) and ends with one less than the length of the sequencethe reason for this is because we began at 0. Figure 6-1 illustrates how sequence items are stored.

Figure 6.1. How sequence elements are stored and accessed



6.1.1. Standard Type Operators

The standard type operators (see Section 4.5) generally work with all sequence types. Of course, one must comparisons with a grain of salt when dealing with objects of mixed types, but the remaining operations will work as advertised.

6.1.2. Sequence Type Operators

A list of all the operators applicable to all sequence types is given in Table 6.1. The operators appear in hierarchical order from highest to lowest with the levels alternating between shaded and not.

Table 6.1. Sequence Type Operators

Sequence Operator	*Function*
`seq[ind]`	Element located at index *ind* of *seq*
`seq[ind1 : ind2]`	Elements from *ind1* up to but not including *ind2* of *seq*
`seq * expr`	*seq* repeated *expr* times
`seq1 + seq2`	Concatenates sequences *seq1* and *seq2*
`obj in seq`	Tests if *obj* is a member of sequence *seq*

obj **not in** *seq*	Tests if *obj* is not a member of sequence *seq*

Membership (`in, not in`)

Membership test operators are used to determine whether an element is *in* or is a member of a sequence. For strings, this test is whether a character is in a string, and for lists and tuples, it is whether an object is an element of those sequences. The **`in`** and **`not in`** operators are Boolean in nature; they return `true` if the membership is confirmed and `False` otherwise.

The syntax for using the membership operators is as follows:

```
obj [not] in sequence
```

Concatenation (`+`)

This operation allows us to take one sequence and join it with another sequence of the same type. The syntax for using the concatenation operator is as follows:

```
sequence1 + sequence2
```

The resulting expression is a new sequence that contains the combined contents of sequences *sequence1* and *sequence2*. Note, however, that although this appears to be the simplest way conceptually to merge the contents of two sequences together, it is not the fastest or most efficient.

For strings, it is less memory-intensive to hold all of the substrings in a list or iterable and use one final `join()` string method call to merge them together. Similarly for lists, it is recommend that readers use the `extend()` list method instead of concatenating two or more lists together. Concatenation comes in handy when you need to merge two sequences together on the fly and cannot rely on mutable object built-in methods that do not have a return value (or more accurately, a return value of `None`). There is an example of this case in the section below on slicing.

Repetition (*)

The repetition operator is useful when consecutive copies of sequence elements are desired. The syntax for using the repetition operator is as follows:

```
sequence * copies_int
```

The number of copies, *copies_int*, must be an integer (prior to 1.6, long integers were not allowed). As with the concatenation operator, the object returned is newly allocated to hold the contents of the multiply replicated objects.

Slices (`[ ], [ : ], [ : : ]`)

To put it simply: *sequences* are data structures that hold objects in an ordered manner. You can get access to individual elements with an index and pair of brackets, or a consecutive group of elements

with the brackets and colons giving the indices of the elements you want starting from one index and going up to but not including the ending index.

Now we are going to explain exactly what we just said in full detail. Sequences are structured data types whose elements are placed sequentially in an ordered manner. This format allows for individual element access by index offset or by an index range of indices to select groups of sequential elements in a sequence. This type of access is called *slicing*, and the slicing operators allow us to perform such access.

The syntax for accessing an individual element is:

```
sequence[index]
```

sequence is the name of the sequence and *index* is the offset into the sequence where the desired element is located. Index values can be positive, ranging from 0 to the maximum index (which is length of the sequence less one). Using the `len()` function (which we will formally introduce in the next section), this gives an index with the range `0 <= index <= len (sequence)-1`.

Alternatively, negative indexes can be used, ranging from -1 to the negative length of the sequence, `-len (sequence)`, i.e., `-len(sequence) <= index <= -1`. The difference between the positive and negative indexes is that positive indexes start from the beginning of the sequences and negative indexes work backward from the end.

Attempting to retrieve a sequence element with an index outside of the length of the sequence results in an `IndexError` exception:

```
>>> names = ('Faye', 'Leanna', 'Daylen')
>>> print names[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: tuple index out of range
```

Because Python is object oriented, you can also directly access an element of a sequence (without first having to assign it to a variable) like this:

```
>>> print ('Faye', 'Leanna', 'Daylen')[1]
Leanna
```

This comes in handy especially in cases where you have called a function and know that you are going to get back a sequence as a return value but are only interested in one or more elements and not the whole thing. So how do we select multiple elements?

Accessing a group of elements is similar to accessing just a single item. Starting and ending indexes may be given, separated by a colon (`:`). The syntax for accessing a group of elements is:

```
sequence[starting_index:ending_index]
```

Using this syntax, we can obtain a "slice" of elements in *sequence* from the *starting_index* up to but not including the element at the *ending_index* index. Both *starting_index* and *ending_index* are optional, and

if not provided, or if `None` is used as an index, the slice will go from the beginning of the sequence or until the end of the sequence, respectively.

In Figures 6-2 to 6-6, we take an entire sequence (of soccer players) of length 5, and explore how to take various slices of such a sequence.

Figure 6-2. Entire sequence: `sequence` or `sequence[:]`

Figure 6-3. Sequence slice: `sequence[0:3]` or `sequence[:3]`

Figure 6-4. Sequence slice: `sequence[2:5]` or `sequence[2:]`

Figure 6-5. Sequence slice: `sequence[1:3]`

Figure 6-6. Sequence slice: `sequence[3]`

Extended Slicing with Stride Indices

The final slice syntax for sequences, known as *extended slicing*, involves a third index known as a *stride*. You can think of a stride index like a "step" value as the third element of a call to the `range()` built-in function or a **`for`** loop in languages like C/C++, Perl, PHP, and Java.

Extended slice syntax with stride indices has actually been around for a long time, built into the Python virtual machine but accessible only via extensions. This syntax was even made available in Jython (and its predecessor JPython) long before version 2.3 of the C interpreter gave everyone else access to it. Here are a few examples:

2.3

Here are a few examples:

```
>>> s = 'abcdefgh'
>>> s[::-1]            # think of it as 'reverse'
'hgfedcba'
>>> s[::2]             # think of it as skipping by 2
'aceg'
```

More on Slice Indexing

The slice index syntax is more flexible than the single element index. The starting and ending indices can exceed the length of the string. In other words, the starting index can start off well left of 0, that is, an index of -100 does not exist, but does not produce an error. Similarly, an index of 100 as an ending index of a sequence with fewer than 100 elements is also okay, as shown here:

```
>>> ('Faye', 'Leanna', 'Daylen')[-100:100]
('Faye', 'Leanna', 'Daylen')
```

Here is another problem: we want to take a string and display it in a loop. Each time through we would like to chop off the last character. Here is a snippet of code that does what we want:

```
>>> s = 'abcde'
>>> i = -1
>>> for i in range(-1, -len(s), -1):
...     print s[:i]
...
abcd
abc
ab
a
```

However, what if we wanted to display the entire string at the first iteration? Is there a way we can do it without adding an additional **`print`** s before our loop? What if we wanted to programmatically specify no index, meaning all the way to the end? There is no real way to do that with an index as we are using negative indices in our example, and `-1` is the "smallest" index. We cannot use `0`, as that would be interpreted as the first element and would not display anything:

```
>>> s[:0]
''
```

Our solution is another tip: using `None` as an index has the same effect as a missing index, so you can get the same functionality programmatically, i.e., when you are using a variable to index through a sequence but also want to be able to access the first or last elements:

```
>>> s = 'abcde'
>>> for i in [None] + range(-1, -len(s), -1):
...     print s[:i]
...
abcde
abcd
abc
ab
a
```

So it works the way we want now. Before parting ways for now, we wanted to point out that this is one of the places where we could have created a list `[None]` and used the `extend()` method to add the `range()` output, or create a list with the `range()` elements and inserted `None` at the beginning, but we are (horribly) trying to save several lines of code here. Mutable object built-in methods like `extend()` do not have a return value, so we could not have used:

```
>>> for i in [None].extend(range(-1, -len(s), -1)):
...     print s[:i]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: iteration over non-sequence
```

The reason for the error is that `[None].extend(...)` returns `None`, which is neither a sequence nor an iterable. The only way we could do it without adding extra lines of code is with the list concatenation

above.

6.1.3. Built-in Functions (BIFs)

Before we look at sequence type BIFs, we wanted to let you know that you will be seeing the term *iterable* mixed in with sequence. The reason for this is that iterables are more generalized and include data types like sequences, iterators, or any object supporting iteration.

Because Python's **`for`** loops can iterate over any iterable type, it will seem like iterating over a pure sequence, even if it isn't one. Also, many of Python's BIFs that previously only accepted sequences as arguments have been upgraded to take iterators and iterator-like objects as well, hence the basket term, "iterable."

We will discuss in detail in this chapter BIFs that have a strong tie to sequences. We will discuss BIFs that apply more specifically to iteration in loops in Chapter 8, "Conditionals and Loops."

Conversion/Casting

The `list()`, `str()`, and `tuple()` BIFs are used to convert from any sequence type to another. You can also think of them as *casting* if coming over from another language, but there really is no conversion or casting going on. These "converters" are really factory functions (introduced in Chapter 4) that take an object and (shallow) copy its contents into a newly generated object of the desired type. Table 6.2 lists the sequence type conversion functions.

Table 6.2. Sequence Type *Conversion* Factory Functions

Function	*Operation*
`list(`*`iter`*`)`	Converts *`iter`*able to a list
`str(`*`obj`*`)`	Converts *`obj`* to string (a printable string representation)
`unicode(`*`obj`*`)`	Converts *`obj`* to a Unicode string (using default encoding)
`basestring()`	Abstract factory function serves only as parent class of `str` and `unicode`, so cannot be called/instantiated (see Section 6.2)
`tuple(`*`iter`*`)`	Converts *`iter`*able to a tuple

Again, we use the term "convert" loosely. But why doesn't Python just convert our argument object into another type? Recall from Chapter 4 that once Python objects are created, we cannot change their identity or their type. If you pass a list to `list()`, a (shallow) copy of the list's objects will be made and inserted into the new list. This is also similar to how the concatenation and repetition operators that we have seen previously do their work.

A shallow copy is where only references are copied...no new objects are made! If you also want copies of the objects (including recursively if you have container objects in containers), you will need to learn about deep copies. More information on shallow and deep copies is available toward the end of this chapter.

The `str()` function is most popular when converting an object into something printable and works with other types of objects, not just sequences. The same thing applies for the Unicode version of `str()`, `unicode()`. The `list()` and `tuple()` functions are useful to convert from one to another (lists to tuples and vice versa). However, although those functions are applicable for strings as well since strings are sequences, using `tuple()` and `list()` to turn strings into tuples or lists (of characters) is not common practice.

Operational

Python provides the following operational BIFs for sequence types (see Table 6.3 below). Note that `len()`, `reversed()`, and `sum()` can only accept sequences while the rest can take iterables. Alternatively, `max()` and `min()` can also take a list of arguments

Table 6.3. Sequence Type *Operational* Built-in Functions

Function	*Operation*
`enumerate(iter)` [a]	Takes an *iter*able and returns an enumerate object (also an iterator) which generates 2-tuple elements (index, item) of *iter* (PEP 279)
`len(seq)`	Returns length (number of items) of *seq*
`max(iter, key=None)` or `max(arg0, arg1..., key=None)` [b]	Returns "largest" element in *iter* or returns "largest" of (*arg0, arg1, ...*); if *key* is present, it should be a callback to pass to the `sort()` method for testing
`min(iter, key=None) or min(arg0, arg1.... key=None)` [b]	Returns "smallest" element in *iter;* returns "smallest" of (*arg0, arg1, ...*); if *key* is present, it should be a callback to pass to the `sort()` method for testing
`reversed(seq)` [c]	Takes *sequence* and returns an iterator that traverses that sequence in reverse order (PEP 322)
`sorted(iter, func=None, key=None, reverse=False)` [c]	Takes an iterable *iter* and returns a sorted list; optional arguments *func, key*, and *reverse* are the same as for the `list.sort()` built-in method
`sum(seq, init=0)` [a]	Returns the sum of the numbers of *seq* and optional *init*ial value; it is equivalent to `reduce (operator.add, seq, init)`
`zip([it0, it1,... itN])` [d]	Returns a list of tuples whose elements are members of each iterable passed into it, i. e., `[(it0[0], it1[0],... itN[0]), (it0[1], it1[1],... itN[1]),... (it0[n], it1[n],... itN[n])]`, where `n` is the minimum cardinality of all of the iterables

[a] New in Python 2.3.

[b] key argument new in Python 2.5.

[c] New in Python 2.4.

[d] New in Python 2.0; more flexibility added in Python 2.4.

We will provide some examples of using these functions with each sequence type in their respective sections.

PREV NEXT

6.2. Strings

Strings are among the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. This contrasts with most other shell-type scripting languages, which use single quotes for literal strings and double quotes to allow escaping of characters. Python uses the "raw string" operator to create literal quotes, so no differentiation is necessary. Other languages such as C use single quotes for characters and double quotes for strings. Python does not have a character type; this is probably another reason why single and double quotes are treated the same.

Nearly every Python application uses strings in one form or another. Strings are a literal or scalar type, meaning they are treated by the interpreter as a singular value and are not containers that hold other Python objects. Strings are immutable, meaning that changing an element of a string requires creating a new string. Strings are made up of individual characters, and such elements of strings may be accessed sequentially via slicing.

With the unification of types and classes in 2.2, there are now actually *three* types of strings in Python. Both regular string (`str`) and Unicode string (`unicode`) types are actually subclassed from an abstract class called `basestring`. This class cannot be instantiated, and if you try to use the factory function to make one, you get this:

```
>>> basestring('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: The basestring type cannot be instantiated
```

How to Create and Assign Strings

Creating strings is as simple as using a scalar value or having the `str()` factory function make one and assigning it to a variable:

```
>>> aString = 'Hello World!'    # using single quotes
>>> anotherString = "Python is cool!" # double quotes
>>> print aString               # print, no quotes!
Hello World!
>>> anotherString               # no print, quotes!
'Python is cool!'

>>> s = str(range(4))           # turn list to string
>>> s
'[0, 1, 2, 3]'
```

How to Access Values (Characters and Substrings) in Strings

Python does not support a character type; these are treated as strings of length one, thus also

considered a substring. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring:

```
>>> aString = 'Hello World!'
>>> aString[0]
'H'
>>> aString[1:5]
'ello'
>>> aString[6:]
'World!'
```

How to Update Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.

```
>>> aString = aString[:6] + 'Python!'
>>> aString
'Hello Python!'
>>> aString = 'different string altogether'
>>> aString
'different string altogether'
```

Like numbers, strings are not mutable, so you cannot change an existing string without creating a new one from scratch. That means that you cannot update individual characters or substrings in a string. However, as you can see above, there is nothing wrong with piecing together parts of your old string into a new string.

How to Remove Characters and Strings

To repeat what we just said, strings are immutable, so you cannot remove individual characters from an existing string. What you can do, however, is to empty the string, or to put together another string that drops the pieces you were not interested in.

Let us say you want to remove one letter from "Hello World!"...the (lowercase) letter "l," for example:

```
>>> aString = 'Hello World!'
>>> aString = aString[:3] + aString[4:]
>>> aString
'Helo World!'
```

To clear or remove a string, you assign an empty string or use the **`del`** statement, respectively:

```
>>> aString = ''
>>> aString
''
>>> del aString
```

In most applications, strings do not need to be explicitly deleted. Rather, the code defining the string

eventually terminates, and the string is eventually deallocated.

PREV NEXT

6.3. Strings and Operators

6.3.1. Standard Type Operators

In Chapter 4, we introduced a number of operators that apply to most objects, including the standard types. We will take a look at how some of those apply to strings. For a brief introduction, here are a few examples using strings:

```
>>> str1 = 'abc'
>>> str2 = 'lmn'
>>> str3 = 'xyz'
>>> str1 < str2
True
>>> str2 != str3
True
>>> str1 < str3 and str2 == 'xyz'
False
```

When using the value comparison operators, strings are compared lexicographically (ASCII value order).

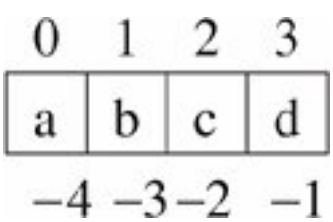
6.3.2. Sequence Operators

Slices ([ ] and [:])

Earlier in Section 6.1.1, we examined how we can access individual or a group of elements from a sequence. We will apply that knowledge to strings in this section. In particular, we will look at:

- Counting forward
- Counting backward
- Default/missing indexes

For the following examples, we use the single string `'abcd'`. Provided in the figure is a list of positive and negative indexes that indicate the position in which each character is located within the string itself.



Using the length operator, we can confirm that its length is 4:

```
>>> aString = 'abcd'
>>> len(aString)
4
```

When counting forward, indexes start at 0 to the left and end at one less than the length of the string (because we started from zero). In our example, the final index of our string is:

```
final_index       = len(aString) - 1
                  = 4 - 1
                  = 3
```

We can access any substring within this range. The slice operator with a single argument will give us a single character, and the slice operator with a range, i.e., using a colon (`:`), will give us multiple consecutive characters. Again, for any ranges `[start:end]`, we will get all characters starting at offset *`start`* up to, but not including, the character at *`end`*. In other words, for all characters *`x`* in the range `[start:end]`, *`start`* `<= x <` *`end`*.

```
 >>> aString[0]
'a'
>>> aString[1:3]
'bc'
>>> aString[2:4]
'cd'
>>> aString[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Any index outside our valid index range (in our example, 0 to 3) results in an error. Above, our access of `aString[2:4]` was valid because that returns characters at indexes 2 and 3, i.e., `'c'` and `'d'`, but a direct access to the character at index 4 was invalid.

When counting backward, we start at index -1 and move toward the beginning of the string, ending at negative value of the length of the string. The final index (the first character) is located at:

```
 final_index     = -len(aString)
                 = -4
>>> aString[-1]
'd'
>>> aString[-3:-1]
'bc'
>>> aString[-4]
'a'
```

When either a starting or an ending index is missing, they default to the beginning or end of the string, respectively.

```
>>> aString[2:]
'cd'
>>> aString[1:]
'bcd'
>>> aString[:-1]
'abc'
>>> aString[:]
'abcd'
```

Notice how the omission of both indices gives us a copy of the entire string.

Membership (`in, not in`)

The membership question asks whether a (sub)string appears in a (nother) string. `true` is returned if that character appears in the string and `False` otherwise. Note that the membership operation is not used to determine if a substring is within a string. Such functionality can be accomplished by using the string methods or string module functions `find()` or `index()` (and their brethren `rfind()` and `rindex()`).

Below are a few more examples of strings and the membership operators. Note that prior to Python 2.3, the **`in`** (and **`not in`**) operators for strings only allowed a single character check, such as the second example below (is '`n`' a substring of '`abcd`'). In 2.3, this was opened up to all strings, not just characters.

```
>>> 'bc' in 'abcd'
True
>>> 'n' in 'abcd'
False
>>> 'nm' not in 'abcd'
True
```

In Example 6.1, we will be using the following predefined strings found in the `string` module:

```
>>> import string
>>> string.uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
```

Example 6.1 is a small script called `idcheck.py` which checks for valid Python identifiers. As we now know, Python identifiers must start with an alphabetic character. Any succeeding characters may be alphanumeric.

Example 6.1. ID Check (`idcheck.py`)

Tests for identifier validity. First symbol must be alphabetic and remaining symbols must be alphanumeric. This tester program only checks identifiers that are at least two characters in length.

```
1  #!usr/bin/env python
2
3  import string
4
5  alphas = string.letters + '_'
6  nums = string.digits
7
8  print 'Welcome to the Identifier Checker v1.0'
9  print 'Testees must be at least 2 chars long.'
10 myInput = raw_input('Identifier to test? ')
11
12 if len(myInput) > 1:
13
14     if myInput[0] not in alphas:
15         print '''invalid: first symbol must be
16             alphabetic'''
17     else:
18         for otherChar in myInput[1:]:
19
20              if otherChar not in alphas + nums:
21                  print '''invalid: remaining
22                     symbols must be alphanumeric'''
23                  break
24         else:
25             print "okay as an identifier"
```

The example also shows use of the string concatenation operator (`+`) introduced later in this section.

Running this script several times produces the following output:

```
$ python idcheck.py
Welcome to the Identifier Checker v1.0
Testees must be at least 2 chars long.
Identifier to test? counter
okay as an identifier
$
$ python idcheck.py
Welcome to the Identifier Checker v1.0
Testees must be at least 2 chars long.
Identifier to test? 3d_effects
invalid: first symbol must be alphabetic
```

Let us take apart the application line by line.

Lines 36

Import the `string` module and use some of the predefined strings to put together valid alphabetic and

numeric identifier strings that we will test against.

Lines 812

Print the salutation and prompt for user input. The **if** statement on line 12 filters out all identifiers or candidates shorter than two characters in length.

Lines 1416

Check to see if the first symbol is alphabetic. If it is not, display the output indicating the result and perform no further processing.

Lines 1718

Otherwise, loop to check the other characters, starting from the second symbol to the end of the string.

Lines 2023

Check to see if each remaining symbol is alphanumeric. Note how we use the concatenation operator (see below) to create the set of valid characters. As soon as we find an invalid character, display the result and perform no further processing by exiting the loop with **break**.

Core Tip: Performance

In general, repeat performances of operations or functions as arguments in a loop are unproductive as far as performance is concerned.

```
while i < len(myString):
        print 'character %d is:', myString[i]
```

The loop above wastes valuable time recalculating the length of string `myString`. This function call occurs for each loop iteration. If we simply save this value once, we can rewrite our loop so that it is more productive.

```
length = len(myString)
while i < length:
    print'character %d is:', myString[i]
```

The same idea applies for this loop above in Example 6.1.

```
for otherChar in myInput[1:]:
     if otherChar not in alphas + nums:
```

*The **for** loop beginning on line 18 contains an **if** statement that concatenates a pair of strings. These strings do not change throughout the course of the application, yet this calculation must be performed*

for each loop iteration. If we save the new string first, we can then reference that string rather than make the same calculations over and over again:

```
alphnums = alphas + nums
for otherChar in myInput[1:]:
    if otherChar not in alphnums:
        :
```

Lines 2425

It may be somewhat premature to show you a **for-else** loop statement, but we are going to give it a shot anyway. (For a full treatment, see Chapter 8). The **else** statement for a **for** loop is optional and, if provided, will execute if the loop finished in completion without being "broken" out of by **break**. In our application, if all remaining symbols check out okay, then we have a valid identifier name. The result is displayed to indicate as such, completing execution.

This application is not without its flaws, however. One problem is that the identifiers tested must have length greater than 1. Our application "as is" is not reflective of the true range of Python identifiers, which may be of length 1. Another problem with our application is that it does not take into consideration Python keywords, which are reserved names that cannot be used for identifiers. We leave these two tasks as exercises for the reader (see Exercise 6-2).

Concatenation (+)

Runtime String Concatenation

We can use the concatenation operator to create new strings from existing ones. We have already seen the concatenation operator in action above in Example 6-1. Here are a few more examples:

```
>>> 'Spanish' + 'Inquisition'
'SpanishInquisition'
>>>
>>> 'Spanish' + ' ' + 'Inquisition'
'Spanish Inquisition'
>>>
>>> s = 'Spanish' + ' ' + 'Inquisition' + ' Made Easy'
>>> s
'Spanish Inquisition Made Easy'
>>>
>>> import string
>>> string.upper(s[:3] + s[20])    # archaic (see below)
'SPAM'
```

The last example illustrates using the concatenation operator to put together a pair of slices from string `s`, the "Spa" from "Spanish" and the "M" from "Made." The extracted slices are concatenated and then sent to the `string.upper()` function to convert the new string to all uppercase letters. String methods were added to Python back in 1.6 so such examples can be replaced with a single call to the final string

method (see example below). There is really no longer a need to import the `string` module unless you are trying to access some of the older string constants which that module defines.

Note: Although easier to learn for beginners, we recommend not using string concatenation when performance counts. The reason is that for every string that is part of a concatenation, Python has to allocate new memory for all strings involved, including the result. Instead, we recommend you either use the string format operator (`%`), as in the examples below, or put all of the substrings in a list, and using one `join()` call to put them all together:

```
>>> '%s %s' % ('Spanish', 'Inquisition')
'Spanish Inquisition'
>>>
>>> s = ' '.join(('Spanish', 'Inquisition', 'Made Easy'))
>>> s
'Spanish Inquisition Made Easy'
>>>
>>> # no need to import string to use string.upper():
>>> ('%s%s' % (s[:3], s[20])).upper()
'SPAM'
```

Compile-Time String Concatenation

The above syntax using the addition operator performs the string concatenation at runtime, and its use is the norm. There is a less frequently used syntax that is more of a programmer convenience feature. Python's syntax allows you to create a single string from multiple string literals placed adjacent to each other in the body of your source code:

```
>>> foo = "Hello" 'world!'
>>> foo
'Helloworld!'
```

It is a convenient way to split up long strings without unnecessary backslash escapes. As you can see from the above, you can mix quotation types on the same line. Another good thing about this feature is that you can add comments too, like this example:

```
>>> f = urllib.urlopen('http://'  # protocol
... 'localhost'                   # hostname
... ':8000'                       # port
... '/cgi-bin/friends2.py')       # file
```

As you can imagine, here is what `urlopen()` really gets as input:

```
>>> 'http://' 'localhost' ':8000' '/cgi-bin/friends2.py'
'http://localhost:8000/cgi-bin/friends2.py'
```

Regular String Coercion to Unicode

When concatenating regular and Unicode strings, regular strings are converted to Unicode first before the operation occurs:

```
>>> 'Hello' + u' ' + 'World' + u'!'
u'Hello World!'
```

Repetition (*)

The repetition operator creates new strings, concatenating multiple copies of the same string to accomplish its functionality:

```
>>> 'Ni!' * 3
'Ni!Ni!Ni!'
>>>
>>> '*'*40
'****************************************'
>>>
>>> print '-' * 20, 'Hello World!', '-' * 20
-------------------- Hello World! --------------------
>>> who = 'knights'
>>> who * 2
'knightsknights'

>>> who
'knights'
```

As with any standard operator, the original variable is unmodified, as indicated in the final dump of the object above.

PREV NEXT

6.4. String-Only Operators

6.4.1. Format Operator (%)

Python features a string format operator. This operator is unique to strings and makes up for the lack of having functions from C's `printf()` family. In fact, it even uses the same symbol, the percent sign (`%`), and supports all the `printf()` formatting codes.

The syntax for using the format operator is as follows:

```
format_string % (arguments_to_convert)
```

The *`format_string`* on the left-hand side is what you would typically find as the first argument to `printf()`: the format string with any of the embedded `%` codes. The set of valid codes is given in Table 6.4. The *`arguments_to_convert`* parameter matches the remaining arguments you would send to `printf()`, namely the set of variables to convert and display.

Table 6.4. Format Operator Conversion Symbols

Format Symbol	*Conversion*
`%c`	Character (integer [ASCII value] or string of length 1)
`%r`[a]	String conversion via `repr()` prior to formatting
`%s`	String conversion via `str()` prior to formatting
`%d / %i`	Signed decimal integer
`%u`[b]	Unsigned decimal integer
`%o`[b]	(Unsigned) octal integer
`%x`[b]`/ %X`	(Unsigned) hexadecimal integer (lower/UPPERcase letters)
`%e / %E`	Exponential notation (with lowercase '`e`'/UPPERcase 'E')
`%f / %F`	Floating point real number (fraction truncates naturally)
`%g / %G`	The shorter of `%e` and `%f/%E%` and `%F%`
`%%`	Percent character (`%`) unescaped

[a] New in Python 2.0; likely unique only to Python.

[b] `%u/%o/%x/%X` of negative int will return a signed string in Python 2.4.

Python supports two formats for the input arguments. The first is a tuple (introduced in Section 2.8, formally in 6.15), which is basically the set of arguments to convert, just like for C's `printf()`. The second format that Python supports is a dictionary (Chapter 7). A dictionary is basically a set of hashed key-value pairs. The keys are requested in the *`format_string`*, and the corresponding values are provided when the string is formatted.

Converted strings can either be used in conjunction with the **`print`** statement to display out to the user or saved into a new string for future processing or displaying to a graphical user interface.

Other supported symbols and functionality are listed in Table 6.5.

Table 6.5. Format Operator Auxiliary Directives

Symbol	*Functionality*
`*`	Argument specifies width or precision
-	Use left justification
+	Use a plus sign (+) for positive numbers
<sp>	Use space-padding for positive numbers
#	Add the octal leading zero ('`0`') or hexadecimal leading '`0x`' or '`0X`', depending on whether '`x`' or '`X`' were used.
`0`	Use zero-padding (instead of spaces) when formatting numbers
`%`	'`%%`' leaves you with a single literal '`%`'
`(var)`	Mapping variable (dictionary arguments)
`m.n`	*`m`* is the minimum total width and *`n`* is the number of digits to display after the decimal point (if applicable)

As with C's `printf()`, the asterisk symbol (`*`) may be used to dynamically indicate the width and precision via a value in argument tuple. Before we get to our examples, one more word of caution: long integers are more than likely too large for conversion to standard integers, so we recommend using exponential notation to get them to fit.

Here are some examples using the string format operator:

Hexadecimal Output

```
>>> "%x" % 108
```

```
'6c'
>>>
>>> "%X" % 108
'6C'
>>>
>>> "%#X" % 108
'0X6C'
>>>
>>> "%#x" % 108
'0x6c'
```

Floating Point and Exponential Notation Output

```
>>>
>>> '%f' % 1234.567890
'1234.567890'
>>>
>>> '%.2f' % 1234.567890
'1234.57'
>>>
>>> '%E' % 1234.567890
'1.234568E+03'
>>>
>>> '%e' % 1234.567890
'1.234568e+03'
>>>
>>> '%g' % 1234.567890
'1234.57'
>>>
>>> '%G' % 1234.567890
'1234.57'
>>>
>>> "%e" % (111111111111111111111L)
'1.111111e+21'
```

Integer and String Output

```
>>> "%+d" % 4
'+4'
>>>

>>> "%+d" % -4
'-4'
>>>
>>> "we are at %d%%" % 100
'we are at 100%'
>>>
>>> 'Your host is: %s' % 'earth'
'Your host is: earth'
>>>
>>> 'Host: %s\tPort: %d' % ('mars', 80)
'Host: mars    Port: 80'
>>>
>>> num = 123
>>> 'dec: %d/oct: %#o/hex: %#X' % (num, num, num)
'dec: 123/oct: 0173/hex: 0X7B'
```

```
>>>
>>> "MM/DD/YY = %02d/%02d/%d" % (2, 15, 67)
'MM/DD/YY = 02/15/67'
>>>
>>> w, p = 'Web', 'page'
>>> 'http://xxx.yyy.zzz/%s/%s.html' % (w, p)
'http://xxx.yyy.zzz/Web/page.html'
```

The previous examples all use tuple arguments for conversion. Below, we show how to use a dictionary argument for the format operator:

```
>>> 'There are %(howmany)d %(lang)s Quotation Symbols' % \
...     {'lang': 'Python', 'howmany': 3}
'There are 3 Python Quotation Symbols'
```

Amazing Debugging Tool

The string format operator is not only a cool, easy-to-use, and familiar feature, but a great and useful debugging tool as well. Practically all Python objects have a string presentation (either evaluatable from `repr()` or `'',` or printable from `str()`). The **`print`** statement automatically invokes the `str()` function for an object. This gets even better. When you are defining your own objects, there are hooks for you to create string representations of your object such that `repr()` and `str()` (and `''` and **`print`**) return an appropriate string as output. And if worse comes to worst and neither `repr()` or `str()` is able to display an object, the Pythonic default is at least to give you something of the format:

```
<... something that is useful ...>
```

6.4.2. String Templates: Simpler Substitution

The string format operator has been a mainstay of Python and will continue to be so. One of its drawbacks, however, is that it is not as intuitive to the new Python programmer not coming from a C/C++ background. Even for current developers using the dictionary form can accidentally leave off the type format symbol, i.e., `%(lang)` vs. the more correct `%(lang)s`. In addition to remembering to put in the correct formatting directive, the programmer must also *know* the type, i.e., is it a string, an integer, etc.

The justification of the new string templates is to do away with having to remember such details and use string substitution much like those in current shell-type scripting languages, the dollar sign (`$`).

The `string` module is temporarily resurrected from the dead as the new `Template` class has been added to it. `Template` objects have two methods, `substitute()` and `safe_substitute()`. The former is more strict, throwing `KeyError` exceptions for missing keys while the latter will keep the substitution string intact when there is a missing key:

```
>>> from string import Template
>>> s = Template('There are ${howmany} ${lang} Quotation Symbols')
>>>
>>> print s.substitute(lang='Python', howmany=3)
There are 3 Python Quotation Symbols
```

```
>>>
>>> print s.substitute(lang='Python')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/usr/local/lib/python2.4/string.py", line 172, in substitute
    return self.pattern.sub(convert, self.template)
  File "/usr/local/lib/python2.4/string.py", line 162, in convert
    val = mapping[named]
KeyError: 'howmany'
>>>
>>> print s.safe_substitute(lang='Python')
There are ${howmany} Python Quotation Symbols
```

The new string templates were added to Python in version 2.4. More information about them can be found in the Python Library Reference Manual and PEP 292.

6.4.3. Raw String Operator (r / R)

The purpose of raw strings, introduced back in version 1.5, is to counteract the behavior of the special escape characters that occur in strings (see the subsection below on what some of these characters are). In raw strings, all characters are taken verbatim with no translation to special or non-printed characters.

This feature makes raw strings absolutely convenient when such behavior is desired, such as when composing regular expressions (see the `re` module documentation). Regular expressions (REs) are strings that define advanced search patterns for strings and usually consist of special symbols to indicate characters, grouping and matching information, variable names, and character classes. The syntax for REs contains enough symbols already, but when you have to insert additional symbols to make special characters act like normal characters, you end up with a virtual "alphanumersymbolic" soup! Raw strings lend a helping hand by not requiring all the normal symbols needed when composing RE patterns.

The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "**r**," which precedes the quotation marks. The "**r**" can be lowercase (**r**) or uppercase (**R**) and must be placed immediately preceding the first quote mark.

In the first of our three examples, we really want a backslash followed by an '`n`' as opposed to a NEWLINE character:

```
>>> '\n'
'\n'
>>> print '\n'

>>> r'\n'
'\\n'
>>> print r'\n'
\n
```

Next, we cannot seem to open our README file. Why not? Because the `\t` and `\r` are taken as special symbols which really are not part of our filename, but are *four* individual characters that are part of our file pathname.

```
>>> f = open('C:\windows\temp\readme.txt', 'r')

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    f = open('C:\windows\temp\readme.txt', 'r')
IOError: [Errno 2] No such file or directory: 'C:\\win-
dows\\temp\readme.txt'
>>> f = open(r'C:\windows\temp\readme.txt', 'r')
>>> f.readline()
'Table of Contents (please check timestamps for last
 update!)\n'
>>> f.close()
```

Finally, we are (ironically) looking for a raw pair of characters `\n` and not NEWLINE. In order to find it, we are attempting to use a simple regular expression that looks for backslash-character pairs that are normally single special whitespace characters:

```
>>> import re
>>> m = re.search('\\[rtfvn]', r'Hello World!\n')
>>> if m is not None: m.group()
...
>>> m = re.search(r'\\[rtfvn]', r'Hello World!\n')
>>> if m is not None: m.group()
...
'\\n'
```

6.4.4. Unicode String Operator (u / U)

The Unicode string operator, uppercase (**U**) and lowercase (**u**), introduced with Unicode string support in Python 1.6, takes standard strings or strings with Unicode characters in them and converts them to a full Unicode string object. More details on Unicode strings are available in Section 6.7.4. In addition, Unicode support is available via string methods (Section 6.6) and the regular expression engine. Here are some examples:

```
u'abc'          U+0061 U+0062 U+0063
u'\u1234'       U+1234
u'abc\u1234\n' U+0061 U+0062 U+0063 U+1234 U+0012
```

The Unicode operator can also accept raw Unicode strings if used in conjunction with the raw string operator discussed in the previous section. The Unicode operator must precede the raw string operator.

```
ur'Hello\nWorld!'
```

6.5. Built-in Functions

6.5.1. Standard Type Functions

cmp()

As with the value comparison operators, the `cmp()` built-in function also performs a lexicographic (ASCII value-based) comparison for strings.

```
>>> str1 = 'abc'
>>> str2 = 'lmn'
>>> str3 = 'xyz'

>>> cmp(str1, str2)
-11
>>> cmp(str3, str1)
23
>>> cmp(str2, 'lmn')
0
```

6.5.2. Sequence Type Functions

len()

```
>>> str1 = 'abc'
>>> len(str1)
3
>>> len('Hello World!')
12
```

The `len()` built-in function returns the number of characters in the string as expected.

max() and min()

```
>>> str2 = 'lmn'
>>> str3 = 'xyz'
>>> max(str2)
'n'
>>> min(str3)
'x'
```

Although more useful with other sequence types, the `max()` and `min()` built-in functions do operate as advertised, returning the greatest and least characters (lexicographic order), respectively. Here are a few more examples:

```
>>> min('ab12cd')
'1'
```

```
>>> min('AB12CD')
'1'
>>> min('ABabCDcd')
'A'
```

enumerate()

```
>>> s = 'foobar'
>>> for i, t in enumerate(s):
...  print i, t
...

0 f
1 o
2 o
3 b
4 a
5 r
```

zip()

```
>>> s, t = 'foa', 'obr'
>>> zip(s, t)
[('f', 'o'), ('o', 'b'), ('a', 'r')]
```

6.5.3. String Type Functions

raw_input()

The built-in `raw_input()` function prompts the user with a given string and accepts and returns a user-input string. Here is an example using `raw_input()`:

```
>>> user_input = raw_input("Enter your name: ")
Enter your name: John Doe
>>>
>>> user_input
'John Doe'
>>>
>>> len(user_input)
8
```

Earlier, we indicated that strings in Python do not have a terminating NUL character like C strings. We added in the extra call to `len()` to show you that what you see is what you get.

str() and unicode()

Both `str()` and `unicode()` are factory functions, meaning that they produce new objects of their type respectively. They will take any object and create a printable or Unicode string representation of the argument object. And, along with `basestring`, they can also be used as arguments along with objects in

`isinstance()` calls to verify type:

```
>>> isinstance(u'\0xAB', str)
False

>>> not isinstance('foo', unicode)
True
>>> isinstance(u'', basestring)
True
>>> not isinstance('foo', basestring)
False
```

`chr()`, `unichr()`, and `ord()`

`chr()` takes a single integer argument in `range(256)` (e.g., between 0 and 255) and returns the corresponding character. `unichr()` does the same thing but for Unicode characters. The range for `unichr()`, added in Python 2.0, is dependent on how your Python was compiled. If it was configured for UCS2 Unicode, then a valid value falls in `range(65536)` or 0x0000-0xFFFF; for UCS4, the value should be in `range(1114112)` or 0x000000-0x110000. If a value does not fall within the allowable range(s), a `ValueError` exception will be raised.

`ord()` is the inverse of `chr()` (for 8-bit ASCII strings) and `unichr()` (for Unicode objects)it takes a single character (string of length 1) and returns the corresponding character with that ASCII code or Unicode code point, respectively. If the given Unicode character exceeds the size specified by your Python configuration, a `TypeError` exception will be thrown.

```
>>> chr(65)
'A'
>>> ord('a')
97
>>> unichr(12345)
u'\u3039'
>>> chr(12345)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    chr(12345)
ValueError: chr() arg not in range(256)
>>> ord(u'\uffffff')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    ord(u'\uffffff')
TypeError: ord() expected a character, but string of
length 2 found
>>> ord(u'\u2345')
9029
```

6.6. String Built-in Methods

String methods were added to Python in the 1.6 to 2.0 timeframethey also were added to Jython. These methods replace most of the functionality in the `string` module as well as to add new functionality. Table 6.6 shows all the current methods for strings. All string methods should fully support Unicode strings. Some are applicable only to Unicode strings.

Table 6.6. String Type Built-in Methods

Method Name	*Description*
string.capitalize()	Capitalizes first letter of *string*
string.center(*width*)	Returns a space-padded *string* with the original *string* centered to a total of *width* columns
string.count(*str*, *beg*= 0, *end*=len(*string*))	Counts how many times *str* occurs in *string*, or in a substring of *string* if starting index *beg* and ending index *end* are given
string.decode(*encoding*='UTF-8', *errors*='strict')	Returns decoded string version of string; on error, default is to raise a `ValueError` unless *errors* is given with '`ignore`' or '`replace`'
string.encode(*encoding*='UTF-8', *errors*='strict') [a]	Returns encoded string version of string; on error, default is to raise a `ValueError` unless *errors* is given with '`ignore`' or '`replace`'
string.endswith(*obj*, *beg*=0, *end*=len(*string*)) [b], [e]	Determines if *string* or a substring of *string* (if starting index *beg* and ending index *end* are given) ends with *obj* where *obj* is typically a string; if *obj* is a tuple, then any of the strings in that tuple; returns `true` if so, and `False` otherwise
string.expandtabs(*tabsize*=8)	Expands tabs in *string* to multiple spaces; defaults to 8 spaces per tab if *tabsize* not provided
string.find(*str*, *beg*=0*end*=len(*string*))	Determine if *str* occurs in *string*, or in a substring of *string* if starting index *beg* and ending index *end* are given; returns index if found and -1 otherwise
string.index(*str*, *beg*=0, *end*=len(*string*))	Same as `find()`, but raises an exception if *str* not found
string.isalnum() [a], [b], [c]	Returns `true` if *string* has at least 1 character and all characters are alphanumeric and `False` otherwise

string.isalpha()[a], [b], [c]	Returns TRue if *string* has at least 1 character and all characters are alphabetic and False otherwise
string.isdecimal()[b], [c], [d]	Returns TRue if *string* contains only decimal digits and False otherwise
string.isdigit()[b], [c]	Returns true if *string* contains only digits and False otherwise
string.islower()[b], [c]	Returns true if *string* has at least 1 cased character and all cased characters are in lowercase and False otherwise
string.isnumeric()[b], [c], [d]	Returns true if *string* contains only numeric characters and False otherwise
string.isspace()[b], [c]	Returns true if *string* contains only whitespace characters and False otherwise
string.istitle()[b], [c]	Returns true if *string* is properly "titlecased" (see title()) and False otherwise
string.isupper()[b], [c]	Returns TRue if *string* has at least one cased character and all cased characters are in uppercase and False otherwise
string.join(*seq*)	Merges (concatenates) the string representations of elements in sequence *seq* into a string, with separator *string*
string.ljust(*width*)	Returns a space-padded *string* with the original string left-justified to a total of *width* columns
string.lower()	Converts all uppercase letters in *string* to lowercase
string.lstrip()	Removes all leading whitespace in *string*
string.partition(*str*)[e]	Like a combination of find() and split(), splits *string* into a 3-tuple (*string_pre_str, str, string_post_str*) on the first occurrence of *str*; if not found, *string_pre_str == string*
string.replace(*str1*, *str2*, *num*=*string*.count(*str1*))	Replaces all occurrences of *str1* in *string* with *str2*, or at most *num* occurrences if *num* given
string.rfind(*str*, *beg*=0, *end*=len(*string*))	Same as find(), but search backward in *string*
string.rindex(*str*, *beg*=0, *end*=len(*string*))	Same as index(), but search backward in *string*
string.rjust(*width*)	Returns a space-padded *string* with the original string right-justified to a total of *width* columns
string.rpartition(*str*)[e]	Same as partition(), but search backwards in *string*
string.rstrip()	Removes all trailing whitespace of *string*

string.split(*str*="", num=*string*.count(*str*))	Splits *string* according to delimiter *str* (space if not provided) and returns list of substrings; split into at most *num* substrings if given
string.splitlines(*num*=string.count('\n'))[b] [c],	Splits *string* at all (or *num*) NEWLINEs and returns a list of each line with NEWLINEs removed
string.startswith(*obj*, *beg*=0, *end*=len(*string*)) [b] [e],	Determines if *string* or a substring of *string* (if starting index *beg* and ending index *end* are given) starts with *obj* where *obj* is typically a string; if *obj* is a tuple, then any of the strings in that tuple; returns `true` if so, and `False` otherwise
string.strip(*[obj]*)	Performs both `lstrip()` and `rstrip()` on *string*
string.swapcase()	Inverts case for all letters in *string*
string.title()[b] [c],	Returns "titlecased" version of *string*, that is, all words begin with uppercase, and the rest are lowercase (also see `istitle()`)
string.translate(*str*, *del*="")	Translates *string* according to translation table *str* (256 chars), removing those in the *del* string
string.upper()	Converts lowercase letters in *string* to uppercase
string.zfill(*width*)	Returns original *string* left-padded with zeros to a total of *width* characters; intended for numbers, `zfill()` retains any sign given (less one zero)

[a] Applicable to Unicode strings only in 1.6, but to all string types in 2.0.

[b] Not available as a `string` module function in 1.5.2.

[e] New or updated in Python 2.5.

[c] New in Jython 2.1.

[d] Applicable to Unicode strings only.

Some examples of using string methods:

```
>>> quest = 'what is your favorite color?'
>>> quest.capitalize()
'What is your favorite color?'
>>>
>>> quest.center(40)
'      what is your favorite color?      '
>>>
>>> quest.count('or')
2
>>>
>>> quest.endswith('blue')
False
```

```
>>>
>>> quest.endswith('color?')
True
>>>
>>> quest.find('or', 30)
-1
>>>
>>> quest.find('or', 22)
25
>>
>>> quest.index('or', 10)
16
>>>
>>> ':'.join(quest.split())
'what:is:your:favorite:color?'
>>> quest.replace('favorite color', 'quest')
>>>
'what is your quest?'
>>>
>>> quest.upper()
'WHAT IS YOUR FAVORITE COLOR?'
```

The most complex example shown above is the one with `split()` and `join()`. We first call `split()` on our string, which, without an argument, will break apart our string using spaces as the delimiter. We then take this list of words and call `join()` to merge our words again, but with a new delimiter, the colon. Notice that we used the `split()` method for our string to turn it into a list, and then, we used the `join()` method for `':'` to merge together the contents of the list.

PREV NEXT

6.7. Special Features of Strings

6.7.1. Special or Control Characters

Like most other high-level or scripting languages, a backslash paired with another single character indicates the presence of a "special" character, usually a nonprintable character, and that this pair of characters will be substituted by the special character. These are the special characters we discussed above that will not be interpreted if the raw string operator precedes a string containing these characters.

In addition to the well-known characters such as NEWLINE (`\n`) and (horizontal) tab (`\t`), specific characters via their ASCII values may be used as well: `\`*`OOO`* or `\x`*`XX`* where *`OOO`* and *`XX`* are their respective octal and hexadecimal ASCII values. Here are the base 10, 8, and 16 representations of 0, 65, and 255:

	ASCII	**ASCII**	**ASCII**
Decimal	`0`	`65`	`255`
Octal	`\000`	`\101`	`\177`
Hexadecimal	`\x00`	`\x41`	`\xFF`

Special characters, including the backslash-escaped ones, can be stored in Python strings just like regular characters.

Another way that strings in Python are different from those in C is that Python strings are not terminated by the NUL (\000) character (ASCII value 0). NUL characters are just like any of the other special backslash-escaped characters. In fact, not only can NUL characters appear in Python strings, but there can be any number of them in a string, not to mention that they can occur anywhere within the string. They are no more special than any of the other control characters. Table 6.7 represents a summary of the escape characters supported by most versions of Python.

Table 6.7. String Literal Backslash Escape Characters

/X	*Oct*	*Dec*	*Hex*	*Char*	*Description*
`\0`	`000`	`0`	`0x00`	`NUL`	Null character
`\a`	`007`	`7`	`0x07`	`BEL`	Bell
`\b`	`010`	`8`	`0x08`	`BS`	Backspace

\t	011	9	0x09	HT	Horizontal tab
\n	012	10	0x0A	LF	Linefeed/Newline
\v	013	11	0x0B	VT	Vertical tab
\f	014	12	0x0C	FF	Form feed
\r	015	13	0x0D	CR	Carriage return
\e	033	27	0x1B	ESC	Escape
\"	042	34	0x22	"	Double quote
\'	047	39	0x27	'	Single quote/apostrophe
\\	134	92	0x5C	\	Backslash

As mentioned before, explicit ASCII octal or hexadecimal values can be given, as well as escaping a NEWLINE to continue a statement to the next line. All valid ASCII character values are between 0 and 255 (octal 0177, hexadecimal 0XFF).

```
\OOO    Octal value OOO (range is 0000 to 0177)
\xXX    'x' plus hexadecimal value XX (range is 0X00 to 0xFF)
\       escape NEWLINE for statement continuation
```

One use of control characters in strings is to serve as delimiters. In database or Internet/Web processing, it is more than likely that most printable characters are allowed as data items, meaning that they would not make good delimiters.

It becomes difficult to ascertain whether or not a character is a delimiter or a data item, and by using a printable character such as a colon (`:`) as a delimiter, you are limiting the number of allowed characters in your data, which may not be desirable.

One popular solution is to employ seldomly used, nonprintable ASCII values as delimiters. These make the perfect delimiters, freeing up the colon and the other printable characters for more important uses.

6.7.2. Triple Quotes

Although strings can be represented by single or double quote delimitation, it is often difficult to manipulate strings containing special or nonprintable characters, especially the NEWLINE character. Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, tabs, and any other special characters.

The syntax for triple quotes consists of three consecutive single or double quotes (used in pairs, naturally):

```
>>> hi = '''hi
there'''
>>> hi             # repr()
'hi\nthere'
>>> print hi       # str()
hi
there
```

Triple quotes lets the developer avoid playing quote and escape character games, all the while bringing at least a small chunk of text closer to WYSIWIG (what you see is what you get) format.

The most powerful use cases are when you have a large block of HTML or SQL that would be completely inconvenient to use by concanentation or wrapped with backslash escapes:

```
errHTML = '''
<HTML><HEAD><TITLE>
Friends CGI Demo</TITLE></HEAD>

<BODY><H3>ERROR</H3>
<B>%s</B><P>
<FORM><INPUT TYPE=button VALUE=Back
ONCLICK="window.history.back()"></FORM>
</BODY></HTML>
'''

cursor.execute('''
         CREATE TABLE users (
         login VARCHAR(8),
         uid INTEGER,
         prid INTEGER)
''')
```

6.7.3. String Immutability

In Section 4.7.2, we discussed how strings are immutable data types, meaning that their values cannot be changed or modified. This means that if you *do* want to update a string, either by taking a substring, concatenating another string on the end, or concatenating the string in question to the end of another string, etc., a new string object must be created for it.

This sounds more complicated than it really is. Since Python manages memory for you, you won't really notice when this occurs. Any time you modify a string or perform any operation that is contrary to immutability, Python will allocate a new string for you. In the following example, Python allocates space for the strings, `'abc'` and `'def'`. But when performing the addition operation to create the string `'abcdef'`, new space is allocated automatically for the new string.

```
>>> 'abc' + 'def'
'abcdef'
```

Assigning values to variables is no different:

```
>>> s = 'abc'
>>> s = s + 'def'
>>> s
'abcdef'
```

In the above example, it looks like we assigned the string '`abc`' to `string`, then appended the string '`def`' to `string`. To the naked eye, strings look mutable. What you cannot see, however, is the fact that a new

string was created when the operation "`s + 'def'`" was performed, and that the new object was then assigned back to `s`. The old string of `'abc'`was deallocated.

Once again, we can use the `id()` built-in function to help show us exactly what happened. If you recall, `id()` returns the "identity" of an object. This value is as close to a "memory address" as we can get in Python.

```
>> s = 'abc'
>>>
>>> id(s)
135060856
>>>
>>> s += 'def'
>>> id(s)
135057968
```

Note how the identities are different for the string before and after the update. Another test of mutability is to try to modify individual characters or substrings of a string. We will now show how any update of a single character or a slice is not allowed:

```
>>> s
'abcdef'
>>>
>>> s[2] = 'C'
Traceback (innermost last):
 File "<stdin>", line 1, in ?
AttributeError: __setitem__
>>>
>>> s[3:6] = 'DEF'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: __setslice__
```

Both operations result in an error. In order to perform the actions that we want, we will have to create new strings using substrings of the existing string, then assign those new strings back to `string`:

```
>>> s
'abcdef'
>>>
>>> s = '%sC%s' % (s[0:2], s[3:])
>>> s
'abCdef'
>>>
>>> s[0:3] + 'DEF'
'abCDEF'
```

So for immutable objects like strings, we make the observation that only valid expressions on the left-hand side of an assignment (to the left of the equals sign [=]) must be the variable representation of an entire object such as a string, not single characters or substrings. There is no such restriction for the expression on the right-hand side.

6.8. Unicode

Unicode string support, introduced to Python in version 1.6, is used to convert between multiple double-byte character formats and encodings, and includes as much functionality as possible to manage these strings. With the addition of string methods (see Section 6.6), Python strings and regular expressions are fully featured to handle a wide variety of applications requiring Unicode string storage, access, and manipulation. We will do our best here to give an overview of Unicode support in Python. But first, let us take a look at some basic terminology and then ask ourselves, just what *is* Unicode?

6.8.1. Terminology

Table 6.8. Unicode Terminology

Term	*Meaning*
ASCII	American Standard Code for Information Interchange
BMP	Basic Multilingual Plane (plane 0)
BOM	Byte Order Mark (character that denotes byte-ordering)
CJK/CJKV	Abbreviation for Chinese-Japanese-Korean (and -Vietnamese)
Code point	Similar to an ASCII value, represents any value in the Unicode codespace, e.g., within `range(1114112)` or integers from 0x000000 to 0x10FFFF.
Octet	Ordered sequence of eight bits as a single unit, aka (8-bit) byte
UCS	Universal Character Set
UCS2	Universal Character Set coded in 2 octets (also see UTF-16)
UCS4	Universal Character Set coded in 4 octets
UTF	Unicode or UCS Transformation Format
UTF-8	8-bit UTF Transformation Format (unsigned byte sequence one to four bytes in length)
UTF-16	16-bit UTF Transformation Format (unsigned byte sequence usually one 16-bit word [two bytes] in length; also see UCS2)

6.8.2. What Is Unicode?

Unicode is the miracle and the mystery that makes it possible for computers to support virtually any language on the planet. Before Unicode, there was ASCII, and ASCII was simple. Every English character was stored in the computer as a seven bit number between 32 and 126. When a user entered the letter A into a text file, the computer would write the letter A to disk as the number 65. Then when the computer opened that file it would translate that number 65 back into an A when it displayed the file contents on the screen.

ASCII files were compact and easy to read. A program could just read in each byte from a file and

convert the numeric value of the byte into the corresponding letter. But ASCII only had enough numbers to represent 95 printable characters. Later software manufacturers extended ASCII to 8 bits, which provided an additional 128 characters, but 223 characters still fell far short of the thousands required to support all non-European languages.

Unicode overcomes the limitations of ASCII by using one or more bytes to represent each character. Using this system, Unicode can currently represent over 90,000 characters.

6.8.3. How Do You Use Unicode?

In the early days, Python could only handle 8-bit ASCII. Strings were simple data types. To manipulate a string, a user had to create a string and then pass it to one of the functions in the `string` module. Then in 2000, we saw the releases of Python 1.6 (and 2.0), the first time Unicode was supported in Python.

In order to make Unicode strings and ASCII strings look as similar as possible, Python strings were changed from being simple data types to real objects. ASCII strings became `StringType`s and Unicode strings became `UnicodeType`s. Both behave very similarly. Both have string methods that correspond to functions in the string module. The `string` module was not updated and remained ASCII only. It is now deprecated and should never be used in any Unicode-compliant code. It remains in Python just to keep legacy code from breaking.

Handling Unicode strings in Python is not that different from handling ordinary ASCII strings. Python calls hard-coded strings string literals. By default all string literals are treated as ASCII. This can be changed by adding the prefix `u` to a string literal. This tells Python that the text inside of the string should be treated as Unicode.

```
>>> "Hello World"   # ASCII string
>>> u"Hello World"  # Unicode string
```

The built-in functions `str()` and `chr()` were not updated to handle Unicode. They only work with regular ASCII strings. If a Unicode string is passed to `str()` it will silently convert the Unicode string to ASCII. If the Unicode string contains any characters that are not supported by ASCII, `str()` will raise an exception. Likewise, `chr()` can only work with numbers 0 to 255. If you pass it a numeric value (of a Unicode character, for example) outside of that range, it will raise an exception.

New BIFs `unicode()` and `unichr()` were added that act just like `str()` and `chr()` but work with Unicode strings. The function `unicode()` can convert any Python data type to a Unicode string and any object to a Unicode representation if that object has an `__unicode__()` method. For a review of these functions, see Sections 6.1.3 and 6.5.3.

6.8.4. What Are Codecs?

The acronym *codec* stands for COder/DECoder. It is a specification for encoding text as byte values and decoding those byte values into text. Unlike ASCII, which used only one byte to encode a character into a number, Unicode uses multiple bytes. Plus Unicode supports several different ways of encoding characters into bytes. Four of the best-known encodings that these codecs can convert are: ASCII, ISO 8859-1/Latin-1, UTF-8, and UTF-16.

The most popular is UTF-8, which uses one byte to encode all the characters in ASCII. This makes it easier for a programmer who has to deal with both ASCII and Unicode text since the numeric values of

the ASCII characters are identical in Unicode.

For other characters, UTF-8 may use one or four bytes to represent a letter, three (mainly) for CJK/East Asian characters, and four for some rare, special use, or historic characters. This makes it more difficult for programmers who have to read and write the raw Unicode data since they cannot just read in a fixed number of bytes for each character. Luckily for us, Python hides all of the details of reading and writing the raw Unicode data for us, so we don't have to worry about the complexities of reading multibyte characters in text streams. All the other codecs are much less popular than UTF-8. In fact, I would say most Python programmers will never have to deal with them, save perhaps UTF-16.

UTF-16 is probably the next most popular codec. It is simpler to read and write its raw data since it encodes every character as a single 16-bit word represented by two bytes. Because of this, the ordering of the two bytes matters. The regular UTF-16 code requires a Byte Order Mark (BOM), or you have to specifically use UTF-16-LE or UTF-16-BE to denote explicit little endian and big endian ordering.

UTF-16 is technically also variable-length like UTF-8 is, but this is uncommon usage. (People generally do not know this or simply do not even care about the rarely used code points in other planes outside the Basic Multilingual Plane (BMP). However, its format is not a superset of ASCII and makes it backward-incompatible with ASCII. Therefore, few programs implement it since most need to support legacy ASCII text.

6.8.5. Encoding and Decoding

Unicode support for multiple codecs means additional hassle for the developer. Each time you write a string to a file, you have to specify the codec (also called an "encoding") that should be used to translate its Unicode characters to bytes. Python minimizes this hassle for us by providing a Unicode string method called `encode()` that reads the characters in the string and outputs the right bytes for the codec we specify.

So every time we write a Unicode string to disk we have to "encode" its characters as a series of bytes using a particular codec. Then the next time we read the bytes from that file, we have to "decode" the bytes into a series of Unicode characters that are stored in a Unicode string object.

Simple Example

The script below creates a Unicode string, encodes it as some bytes using the UTF-8 codec, and saves it to a file. Then it reads the bytes back in from disk and decodes them into a Unicode string. Finally, it prints the Unicode string so we can see that the program worked correctly.

Line-by-Line Explanation

Lines 17

The usual setup plus a doc string and some constants for the codec we are using and the name of the file we are going to store the string in.

Lines 919

Here we create a Unicode string literal, encode it with our codec, and write it out to disk (lines 9-13). Next, we read the data back in from the file, decode it, and display it to the screen, suppressing the **`print`** statement's NEWLINE because we are using the one saved with the string (lines 15-19).

Example 6.2. Simple Unicode String Example (uniFile.py)

This simple script writes a Unicode string to disk and reads it back in for display. It encodes it into UTF-8 for writing to disk, which it must then decode in to display it.

```
#!/usr/bin/env python
'''
An example of reading and writing Unicode strings: Writes
a Unicode string to a file in utf-8 and reads it back in.
'''
CODEC = 'utf-8'
FILE = 'unicode.txt'

hello_out = u"Hello world\n"
bytes_out = hello_out.encode(CODEC)
f = open(FILE, "w")
f.write(bytes_out)
f.close()

f = open(FILE, "r")
bytes_in = f.read()
f.close()
hello_in = bytes_in.decode(CODEC)
print hello_in,
```

When we run the program we get the following output:

```
$ unicode_example.py
Hello World
```

We also find a file called unicode.txt on the file system that contains the same string the program printed out.

```
$ cat unicode.txt
Hello World!
```

Simple Web Example

We show a similar and simple example of using Unicode with CGI in the Web Programming chapter (Chapter 20).

6.8.6. Using Unicode in Real Life

Examples like this make it look deceptively easy to handle Unicode in your code, and it is pretty easy, as long as you follow these simple rules:

- Always prefix your string literals with `u`.
- Never use `str()`... always use `unicode()` instead.

- Never use the outdated `string` moduleit blows up when you pass it any non-ASCII characters.
- Avoid unnecessary encoding and decode of Unicode strings in your program. Only call the `encode()` method right before you write your text to a file, database, or the network, and only call the `decode()` method when you are reading it back in.

These rules will prevent 90 percent of the bugs that can occur when handling Unicode text. The problem is that the other 10 percent of the bugs are beyond your control. The greatest strength of Python is the huge library of modules that exist for it. They allow Python programmers to write a program in ten lines of code that might require a hundred lines of code in another language. But the quality of Unicode support within these modules varies widely from module to module.

Most of the modules in the standard Python library are Unicode compliant. The biggest exception is the `pickle` module. Pickling only works with ASCII strings. If you pass it a Unicode string to unpickle, it will raise an exception. You have to convert your string to ASCII first. It is best to avoid using text-based pickles. Fortunately, the binary format is now the default and it is better to stick with it. This is especially true if you are storing your pickles in a database. It is much better to save them as a BLOB than to save them as a TEXT or VARCHAR field and then have your pickles get corrupted when someone changes your column type to Unicode.

If your program uses a bunch of third-party modules, then you will probably run into a number of frustrations as you try to get all of the programs to speak Unicode to each other. Unicode tends to be an all-or-nothing proposition. Each module in your system (and all systems your program interfaces with) has to use Unicode and the same Unicode codec. If any one of these systems does not speak Unicode, you may not be able to read and save strings properly.

As an example, suppose you are building a database-enabled Web application that reads and writes Unicode. In order to support Unicode you need the following pieces to all support Unicode:

- Database server (MySQL, PostgreSQL, SQL Server, etc.)
- Database adapter (`MySQLdb`, etc.)
- Web framework (`mod_python`, `cgi`, Zope, Plane, Django etc.)

The database server is often the easiest part. You just have to make sure that all of your tables use the UTF-8 encoding.

The database adapter can be trickier. Some database adapters support Unicode, some do not. `MySQLdb`, for instance, does not default to Unicode mode. You have to use a special keyword argument `use_unicode` in the `connect()` method to get Unicode strings in the result sets of your queries.

Enabling Unicode is very simple to do in `mod_python`. Just set the text-encoding field to `"utf-8"` on the request object and `mod_python` handles the rest. Zope and other more complex systems may require more work.

6.8.7. Real-Life Lessons Learned

Mistake #1: You have a large application to write under significant time pressure. Foreign language support was a requirement, but no specifics are made available by the product manager. You put off Unicode-compliance until the project is mostly complete ... it is not going to be that much effort to add Unicode support anyway, right?

Result #1: Failure to anticipate the foreign-language needs of end-users as well as integration of Unicode support with the other foreign language-oriented applications that they used. The retrofit of the entire system would be extremely tedious and time-consuming.

Mistake #2: Using the `string` module everywhere including calling `str()` and `chr()` in many places throughout the code.

Result #2: Convert to string methods followed by global search-and-replace of `str()` and `chr()` with `unicode()` and `unichr()`. The latter breaks all pickling. The pickling format has to be changed to binary. This in turn breaks the database schema, which needs to be completely redone.

Mistake #3: Not confirming that all auxiliary systems support Unicode fully.

Result #3: Having to patch those other systems, some of which may not be under your source control. Fixing Unicode bugs everywhere leads to code instability and the distinct possibility of introducing new bugs.

Summary: Enabling full Unicode and foreign-language compliance of your application is a project on its own. It needs to be well thought out and planned carefully. All software and systems involved must be "checked off," including the list of Python standard library and/or third-party external modules that are to be used. You may even have to bring onboard an entire team with internationalization (or "I18N") experience.

6.8.8. Unicode Support in Python

`unicode()` Built-in Function

The Unicode factory function should operate in a manner similar to that of the Unicode string operator (`u` / `U`). It takes a string and returns a Unicode string.

`decode()/encode()` Built-in Methods

The `decode()` and `encode()` built-in methods take a string and return an equivalent decoded/encoded string. `decode()` and `encode()` work for both regular and Unicode strings. `decode()` was added to Python in 2.2.

Unicode Type

A Unicode string object is subclassed from basestring and an instance is created by using the `unicode()` factory function, or by placing a `u` or `U` in front of the quotes of a string. Raw strings are also supported. Prepend a `ur` or `UR` to your string literal.

Unicode Ordinals

The standard `ord()` built-in function should work the same way. It was enhanced recently to support Unicode objects. The `unichr()` built-in function returns a Unicode object for a character (provided it is a 32-bit value); otherwise, a `ValueError` exception is raised.

Coercion

Mixed-mode string operations require standard strings to be converted to Unicode objects.

Exceptions

`UnicodeError` is defined in the exceptions module as a subclass of `ValueError`. All exceptions related to Unicode encoding/decoding should be subclasses of `UnicodeError`. See also the string `encode()` method.

Standard Encodings

Table 6.9 presents an extremely short list of the more common encodings used in Python. For a more complete listing, please see the Python Documentation. Here is an online link:

http://docs.python.org/lib/standard-encodings.html

RE Engine Unicode-Aware

The regular expression engine should be Unicode aware. See the `re` Code Module sidebar in Section 6.9.

Table 6.9. Common Unicode Codecs/Encodings

Codec	*Description*
`utf-8`	8-bit variable length encoding (default encoding)
`utf-16`	16-bit variable length encoding (little/big endian)
`utf-16-le`	UTF-16 but explicitly little endian
`utf-16-be`	UTF-16 but explicitly big endian
`ascii`	7-bit ASCII codepage
`iso-8859-1`	ISO 8859-1 (Latin-1) codepage
`unicode-escape`	(See Python Unicode Constructors for a definition)
`raw-unicode-escape`	(See Python Unicode Constructors for a definition)
`native`	Dump of the internal format used by Python

String Format Operator

For Python format strings: `%s` performs `str(u)` for Unicode objects embedded in Python strings, so the output will be `u.encode(<default encoding>)`. If the format string is a Unicode object, all parameters are coerced to Unicode first and then put together and formatted according to the format string. Numbers are first converted to strings and then to Unicode. Python strings are interpreted as Unicode strings using the <default encoding>. Unicode objects are taken as is. All other string formatters should work accordingly. Here is an example:

`u"%s %s" % (u"abc", "abc")` ⟹ `u"abc abc"`

6.9. Related Modules

Table 6.10 lists the key related modules for strings that are part of the Python standard library.

Table 6.10. Related Modules for String Types

Module	*Description*
`string`	String manipulation and utility functions, i.e., Template class
`re`	Regular expressions: powerful string pattern matching
`struct`	Convert strings to/from binary data format
`c/StringIO`	String buffer object that behaves like a file
`base64`	Base 16, 32, and 64 data encoding and decoding
`codecs`	Codec registry and base classes
`crypt`	Performs one-way encryption cipher
`difflib`[a]	Various "differs" for sequences
`hashlib`[b]	API to many different secure hash and message digest algorithms
`hmac`[c]	Keyed-hashing for message authentication
`md5`[d]	RSA's MD5 message digest authentication
`rotor`	Provides multi-platform en/decryption services
`sha`[d]	NIST's secure hash algorithm SHA
`stringprep`[e]	Prepares Unicode strings for use in Internet protocols
`textwrap`[e]	Text-wrapping and filling
`unicodedata`	Unicode database

[a] New in Python 2.1.

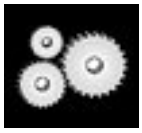
[b] New in Python 2.5.

[c] New in Python 2.2.

[d] Obsoleted in Python 2.5 by `hashlib` module.

[e] New in Python 2.3.

Core Module: `re`



Regular expressions (REs) provide advanced pattern matching scheme for strings. Using a separate syntax that describes these patterns, you can effectively use them as "filters" when passing in the text to perform the searches on. These filters allow you to extract the matched patterns as well as perform find-and-replace or divide up strings based on the patterns that you describe.

The `re` module, introduced in Python 1.5, obsoletes the original `regex` and `regsub` modules from earlier releases. It represented a major upgrade in terms of Python's support for regular expressions, adopting the complete Perl syntax for REs. In Python 1.6, a completely new engine was written (SRE), which added support for Unicode strings as well as significant performance improvements. SRE replaces the old PCRE engine, which had been under the covers of the regular expression modules.

Some of the key functions in the `re` module include: `compile()` compiles an RE expression into a reusable RE object; `match()` attempts to match a pattern from the beginning of a string; `search()` searches for any matching pattern in the string; and `sub()` performs a search-and-replace of matches. Some of these functions return match objects with which you can access saved group matches (if any were found). All of Chapter 15 is dedicated to regular expressions.

6.10. Summary of String Highlights

Characters Delimited by Quotation Marks

You can think of a string as a Python data type that you can consider as an array or contiguous set of characters between any pair of Python quotation symbols, or quotes. The two most common quote symbols for Python are the single quote, a single forward apostrophe ('), and the double quotation mark ("). The actual string itself consists entirely of those characters in between and not the quote marks themselves.

Having the choice between two different quotation marks is advantageous because it allows one type of quote to serve as a string delimiter while the other can be used as characters within the string without the need for special escape characters. Strings enclosed in single quotes may contain double quotes as characters and vice versa.

No Separate Character Type

Strings are the only literal sequence type, a sequence of characters. However, characters are not a type, so strings are the lowest-level primitive for character storage and manipulation. Characters are simply strings of length one.

String Format Operator (%) Provides `printf()`-like Functionality

The string format operator (see Section 6.4.1) provides a flexible way to create a custom string based on variable input types. It also serves as a familiar interface to formatting data for those coming from the C/C++ world.

Triple Quotes

In Section 6.7.2, we introduced the notion of triple quotes, which are strings that can have special embedded characters like NEWLINEs and tabs. Triple-quoted strings are delimited by pairs of three single (`' ' '`) or double (`" " "`) quotation marks.

Raw Strings Takes Special Characters Verbatim

In Section 6.4.2, we introduced raw strings and discussed how they do not interpret special characters escaped with the backslash. This makes raw strings ideal for situations where strings must be taken verbatim, for example, when describing regular expressions.

Python Strings Do *Not* End with NUL or '\0'

One major problem in C is running off the end of a string into memory that does not belong to you. This occurs when strings in C are not properly terminated with the NUL or '\0' character (ASCII value of zero). Along with managing memory for you, Python also removes this little burden or annoyance. Strings in Python do not terminate with NUL, and you do not have to worry about adding them on. Strings consist entirely of the characters that were designated and nothing more.

6.11. Lists

Like strings, lists provide sequential storage through an index offset and access to single or consecutive elements through slices. However, the comparisons usually end there. Strings consist only of characters and are immutable (cannot change individual elements), while lists are flexible container objects that hold an arbitrary number of Python objects. Creating lists is simple; adding to lists is easy, too, as we see in the following examples.

The objects that you can place in a list can include standard types and objects as well as user-defined ones. Lists can contain different types of objects and are more flexible than an array of C structs or Python arrays (available through the external array module) because arrays are restricted to containing objects of a single type. Lists can be populated, empty, sorted, and reversed. Lists can be grown and shrunk. They can be taken apart and put together with other lists. Individual or multiple items can be inserted, updated, or removed at will.

Tuples share many of the same characteristics of lists and although we have a separate section on tuples, many of the examples and list functions are applicable to tuples as well. The key difference is that tuples are immutable, i.e., read-only, so any operators or functions that allow updating lists, such as using the slice operator on the left-hand side of an assignment, will not be valid for tuples.

How to Create and Assign Lists

Creating lists is as simple as assigning a value to a variable. You handcraft a list (empty or with elements) and perform the assignment. Lists are delimited by surrounding square brackets (**[ ]**). You can also use the factory function.

```
>>> aList = [123, 'abc', 4.56, ['inner', 'list'], 7-9j]
>>> anotherList = [None, 'something to see here']
>>> print aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
>>> print anotherList
[None, 'something to see here']
>>> aListThatStartedEmpty = []
>>> print aListThatStartedEmpty
[]
>>> list('foo')
['f', 'o', 'o']
```

How to Access Values in Lists

Slicing works similar to strings; use the square bracket slice operator (**[ ]**) along with the index or indices.

```
>>> aList[0]
123
>>> aList[1:4]
['abc', 4.56, ['inner', 'list']]
>>> aList[:3]
[123, 'abc', 4.56]
>>> aList[3][1]
'list'
```

How to Update Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the `append()` method:

```
>>> aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
>>> aList[2]
4.56
>>> aList[2] = 'float replacer'
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>>
>>> anotherList.append("hi, i'm new here")
>>> print anotherList
[None, 'something to see here', "hi, i'm new here"]
>>> aListThatStartedEmpty.append('not empty anymore')
>>> print aListThatStartedEmpty
['not empty anymore']
```

How to Remove List Elements and Lists

To remove a list element, you can use either the **`del`** statement if you know exactly which element(s) you are deleting or the `remove()` method if you do not know.

```
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>> del aList[1]
>>> aList
[123, 'float replacer', ['inner', 'list'], (7-9j)]
>>> aList.remove(123)
>>> aList
['float replacer', ['inner', 'list'], (7-9j)]
```

You can also use the `pop()` method to remove and return a specific object from a list.

Normally, removing an entire list is not something application programmers do. Rather, they tend to let it go out of scope (i.e., program termination, function call completion, etc.) and be deallocated, but if they do want to explicitly remove an entire list, they use the **`del`** statement:

```
del aList
```

6.12. Operators

6.12.1. Standard Type Operators

In Chapter 4, we introduced a number of operators that apply to most objects, including the standard types. We will take a look at how some of those apply to lists.

```
>>> list1 = ['abc', 123]
>>> list2 = ['xyz', 789]
>>> list3 = ['abc', 123]
>>> 1ist1 < list2
True
>>> list2 < list3
False
>>> list2 > list3 and list1 == list3
True
```

When using the value comparison operators, comparing numbers and strings is straightforward, but not so much for lists, however. List comparisons are somewhat tricky, but logical. The comparison operators use the same algorithm as the `cmp()` built-in function. The algorithm basically works like this: the elements of both lists are compared until there is a determination of a winner. For example, in our example above, the output of `'abc'` versus `'xyz'` is determined immediately, with `'abc' < 'xyz'`, resulting in `list1 < list2` and `list2 >= list3`. Tuple comparisons are performed in the same manner as lists.

6.12.2. Sequence Type Operators

Slices ([ ] and [:])

Slicing with lists is very similar to strings, but rather than using individual characters or substrings, slices of lists pull out an object or a group of objects that are elements of the list operated on. Focusing specifically on lists, we make the following definitions:

```
>>> num_list = [43, -1.23, -2, 6.19e5]
>>> str_list = ['jack', 'jumped', 'over', 'candlestick']
>>> mixup_list = [4.0, [1, 'x'], 'beef', -1.9+6j]
```

Slicing operators obey the same rules regarding positive and negative indexes, starting and ending indexes, as well as missing indexes, which default to the beginning or to the end of a sequence.

```
>>> num_list[1]
-1.23
>>>
>>> num_list[1:]
[-1.23, -2, 619000.0]
>>>
>>> num_list[2:-1]
[-2]
>>>
```

```
>>> str_list[2]
'over'
>>> str_list[:2]
['jack', 'jumped']
>>>
>>> mixup_list
[4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>> mixup_list[1]
[1, 'x']
```

Unlike strings, an element of a list might also be a sequence, implying that you can perform all the sequence operations or execute any sequence built-in functions on that element. In the example below, we show that not only can we take a slice of a slice, but we can also change it, and even to an object of a different type. You will also notice the similarity to multidimensional arrays.

```
>>> mixup_list[1][1]
'x'
>>> mixup_list[1][1] = -64.875
>>> mixup_list
[4.0, [1, -64.875], 'beef', (-1.9+6j)]
```

Here is another example using `num_list`:

```
>>> num_list
[43, -1.23, -2, 6.19e5]
>>>
>>> num_list[2:4] = [16.0, -49]
>>>
>>> num_list
[43, -1.23, 16.0, -49]
>>>
>>> num_list[0] = [65535L, 2e30, 76.45-1.3j]
>>>
>>> num_list
[[65535L, 2e+30, (76.45-1.3j)], -1.23, 16.0, -49]
```

Notice how, in the last example, we replaced only a single element of the list, but we replaced it with a list. So as you can tell, removing, adding, and replacing things in lists are pretty freeform. Keep in mind that in order to splice elements of a list into another list, you have to make sure that the left-hand side of the assignment operator (`=`) is a slice, not just a single element.

Membership (`in`, `not in`)

With lists (and tuples), we can check whether an object is a member of a list (or tuple).

```
>>> mixup_list
[4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>>
>>> 'beef' in mixup_list
True
>>>
>>> 'x' in mixup_list
```

```
False
>>>
>>> 'x' in mixup_list[1]
True
>>> num_list

[[65535L, 2e+030, (76.45-1.3j)], -1.23, 16.0, -49]
>>>
>>> -49 in num_list
True
>>>
>>> 34 in num_list
False
>>>
>>> [65535L, 2e+030, (76.45-1.3j)] in num_list
True
```

Note how `'x'` is *not* a member of `mixup_list`. That is because `'x'` itself is not actually a member of `mixup_list`. Rather, it is a member of `mixup_uplist[1]`, which itself is a list. The membership operator is applicable in the same manner for tuples.

Concatenation (+)

The concatenation operator allows us to join multiple lists together. Note in the examples below that there is a restriction of concatenating like objects. In other words, you can concatenate only objects of the same type. You cannot concatenate two different types even if both are sequences.

```
>>> num_list = [43, -1.23, -2, 6.19e5]
>>> str_list = ['jack', 'jumped', 'over', 'candlestick']
>>> mixup_list = [4.0, [1, 'x'], 'beef', -1.9+6j]
>>>
>>> num_list + mixup_list
[43, -1.23, -2, 619000.0, 4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>>
>>> str_list + num_list
['jack', 'jumped', 'over', 'candlestick', 43, -1.23, -2, 619000.0]
```

As we will discover in Section 6.13, starting in Python 1.5.2, you can use the `extend()` method in place of the concatenation operator to append the contents of a list to another. Using `extend()` is advantageous over concatenation because it actually appends the elements of the new list to the original, rather than creating a new list from scratch like `+` does. `extend()` is also the method used by the augmented assignment or in-place concatenation operator (`+=`), which debuted in Python 2.0.

We would also like to point out that the concatenation operator *does not* facilitate adding individual elements to a list. The upcoming example illustrates a case where attempting to add a new item to the list results in failure.

```
>>> num_list + 'new item'
```

```
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

This example fails because we had different types to the left and right of the concatenation operator. A combination of (list `+` string) is not valid. Obviously, our intention was to add the `'new item'` string to the list, but we did not go about it the proper way. Fortunately, we have a solution:

Use the `append()` list built-in method (we will formally introduce `append()` and all other built-in methods in Section 6.13):

```
>>> num_list.append('new item')
```

Repetition (*)

Use of the repetition operator may make more sense with strings, but as a sequence type, lists and tuples can also benefit from this operation, if needed:

```
>>> num_list * 2
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0]
>>>
>>> num_list * 3
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0, 43,
-1.23, -2, 619000.0]
```

Augmented assignment also works, beginning in Python 2.0:

```
>>> hr = '-'
>>> hr *= 30
>>> hr
'------------------------------'
```

6.12.3. List Type Operators and List Comprehensions

There are really no special list-only operators in Python. Lists can be used with most object and sequence operators. In addition, list objects have their own methods. One construct that lists *do* have however, are *list comprehensions*. These are a combination of using list square brackets and a **`for`**-loop inside, a piece of logic that dictates the contents of the list object to be created. We cover list comprehensions in Chapter 8, but we present a simple example here as well as a few more throughout the remainder of the the chapter:

```
>>> [ i * 2 for i in [8, -2, 5] ]
[16, -4, 10]
>>> [ i for i in range(8) if i % 2 == 0 ]
[0, 2, 4, 6]
```

6.13. Built-in Functions

6.13.1. Standard Type Functions

cmp()

In Section 4.6.1, we introduced the `cmp()` built-in function with examples of comparing numbers and strings. But how would `cmp()` work with other objects such as lists and tuples, which can contain not only numbers and strings, but other objects like lists, tuples, dictionaries, and even user-created objects?

```
>>> list1, list2 = [123, 'xyz'], [456, 'abc']
>>> cmp(list1, list2)
-1
>>>
>>> cmp(list2, list1)
1
>>> list3 = list2 + [789]
>>> list3
[456, 'abc', 789]
>>>
>>> cmp(list2, list3)
-1
```

Compares are straightforward if we are comparing two objects of the same type. For numbers and strings, the direct values are compared, which is trivial. For sequence types, comparisons are somewhat more complex, but similar in manner. Python tries its best to make a fair comparison when one cannot be made, i.e., when there is no relationship between the objects or when types do not even have compare functions, then all bets are off as far as obtaining a "logical" decision.

Before such a drastic state is arrived at, more safe-and-sane ways to determine an inequality are attempted. How does the algorithm start? As we mentioned briefly above, elements of lists are iterated over. If these elements are of the same type, the standard compare for that type is performed. As soon as an inequality is determined in an element compare, that result becomes the result of the list compare. Again, these element compares are for elements of the same type. As we explained earlier, when the objects are different, performing an accurate or true comparison becomes a risky proposition.

When we compare `list1` with `list2`, both lists are iterated over. The first true comparison takes place between the first elements of both lists, i.e., `123` vs. `456`. Since `123` < `456`, `list1` is deemed "smaller."

If both values are the same, then iteration through the sequences continues until either a mismatch is found, or the end of the shorter sequence is reached. In the latter case, the sequence with more elements is deemed "greater." That is the reason why we arrived above at `list2 < list3`. Tuples are compared using the same algorithm. We leave this section with a summary of the algorithm highlights:

1. Compare elements of both lists.
2.

If elements are of the same type, perform the compare and return the result.

3.

If elements are different types, check to see if they are numbers.

a.

If numbers, perform numeric coercion if necessary and compare.

b.

If either element is a number, then the other element is "larger" (numbers are "smallest").

c.

Otherwise, types are sorted alphabetically by name.

4.

If we reach the end of one of the lists, the longer list is "larger."

5.

If we exhaust both lists and share the same data, the result is a tie, meaning that 0 is returned.

6.13.2. Sequence Type Functions

`len()`

For strings, `len()` gives the total length of the string, as in the number of characters. For lists (and tuples), it will not surprise you that `len()` returns the number of elements in the list (or tuple). Container objects found within count as a single item. Our examples below use some of the lists already defined above in previous sections.

```
>>> len(num_list)
4
>>>
>>> len(num_list*2)
8
```

`max()` and `min()`

`max()` and `min()` did not have a significant amount of usage for strings since all they did was to find the "largest" and "smallest" characters (lexicographically) in the string. For lists (and tuples), their functionality is more defined. Given a list of like objects, i.e., numbers or strings only, `max()` and `min()` could come in quite handy. Again, the quality of return values diminishes as mixed objects come into play. However, more often than not, you will be using these functions in a situation where they will provide the results you are seeking. We present a few examples using some of our earlier-defined lists.

```
>>> max(str_list)
'park'
>>> max(num_list)
[65535L, 2e+30, (76.45-1.3j)]
>>> min(str_list)
'candlestick'
>>> min(num_list)
-49
```

`sorted()` and `reversed()`

```
>>> s = ['They', 'stamp', 'them', 'when', "they're", 'small']
>>> for t in reversed(s):
...  print t,
...
small they're when them stamp They
>>> sorted(s)
['They', 'small', 'stamp', 'them', "they're", 'when']
```

For beginners using strings, notice how we are able to mix single and double quotes together in harmony with the contraction "they're." Also to those new to strings, this is a note reminding you that all string sorting is lexicographic and not alphabetic (the letter "T" comes before the letter "a" in the ASCII table.)

`enumerate()` and `zip()`

```
>>> albums = ['tales', 'robot', 'pyramid']
>>> for i, album in enumerate(albums):
...     print i, album
...
0 tales
1 robot
2 pyramid
>>>
>>> fn = ['ian', 'stuart', 'david']
>>> ln = ['bairnson', 'elliott', 'paton']
>>>
>>> for i, j in zip(fn, ln):
...     print ('%s %s' % (i,j)).title()
...
Ian Bairnson
Stuart Elliott
David Paton
```

`sum()`

```
>>> a = [6, 4, 5]
>>> reduce(operator.add, a)
15
>>> sum(a)
15
```

```
>>> sum(a, 5)
20
>>> a = [6., 4., 5.]
>>> sum(a)
15.0
```

`list()` and `tuple()`

The `list()` and `tuple()` factory functions take iterables like other sequences and make new lists and tuples, respectively, out of the (just shallow-copied) data. Although strings are also sequence types, they are not commonly used with `list()` and `tuple()`. These built-in functions are used more often to convert from one type to the other, i.e., when you have a tuple that you need to make a list (so that you can modify its elements) and vice versa.

```
>>> aList = ['tao', 93, 99, 'time']
>>> aTuple = tuple(aList)
>>> aList, aTuple
(['tao', 93, 99, 'time'], ('tao', 93, 99, 'time'))
>>> aList == aTuple
False
>>> anotherList = list(aTuple)
>>> aList == anotherList
True
>>> aList is anotherList
False
>>> [id(x) for x in aList, aTuple, anotherList]
[10903800, 11794448, 11721544]
```

As we already discussed at the beginning of the chapter, neither `list()` nor `tuple()` performs true *conversions* (see also Section 6.1.2). In other words, the list you passed to `tuple()` does not turn into a list, and the tuple you give to `list()` does not really become a list. Although the data set for both (the original and new object) is the same (hence satisfying `==`), neither variable points to the same object (thus failing `is`). Also notice that, even though their values are the same, a list cannot "equal" a tuple.

6.13.3. List Type Built-in Functions

There are currently no special list-only built-in functions in Python unless you consider `range()` as oneits sole function is to take numeric input and generate a list that matches the criteria. `range()` is covered in Chapter 8. Lists can be used with most object and sequence built-in functions. In addition, list objects have their own methods.

6.14. List Type Built-in Methods

Lists in Python have *methods*. We will go over methods more formally in an introduction to object-oriented programming in Chapter 13, but for now think of methods as functions or procedures that apply only to specific objects. So the methods described in this section behave just like built-in functions except that they operate only on lists. Since these functions involve the mutability (or updating) of lists, none of them is applicable for tuples.

You may recall our earlier discussion of accessing object attributes using the dotted attribute notation: *object.attribute*. List methods are no different, using *list.method()*. We use the dotted notation to access the attribute (here it is a function), then use the function operators (**( )**) in a functional notation to invoke the methods.

We can use `dir()` on a list object to get its attributes including its methods:

```
>>> dir(list)    # or dir([])
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__delslice__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__',
'__getslice__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__',
'__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__setslice__', '__str__',
'append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

Table 6.11 shows all the methods currently available for lists. Some examples of using various list methods are shown later.

Table 6.11. List Type Built-in Methods

List Method	*Operation*
list.append(*obj*)	Adds *obj* to the end of *list*
list.count(*obj*)	Returns count of how many times *obj* occurs in *list*
list.extend(*seq*)[a]	Appends contents of *seq* to *list*
list.index(*obj, i=0, j=len(list)*)	Returns lowest index *k* where *list[k]==obj* and *i<= k<j*; otherwise `ValueError` raised
list.insert(*index, obj*)	Inserts *obj* into *list* at offset *index*

list.pop(*index*=-1)[a]	Removes and returns *obj* at given or last *index* from *list*
list.remove(*obj*)	Removes object *obj* from *list*
list.reverse()	Reverses objects of *list* in place
list.sort(*func*=None, *key*=None, *reverse*=False)[b]	Sorts list members with optional comparison *func*tion; *key* is a callback when extracting elements for sorting, and if *reverse* flag is `true`, then list is sorted in reverse order

[a] New in Python 1.5.2.

[b] Support for `key` and `reverse` added in Python 2.4.

```
>>> music_media = [45]
>>> music_media
[45]
>>>
>>> music_media.insert(0, 'compact disc')
>>> music_media
['compact disc', 45]
>>>
>>> music_media.append('long playing record')
>>> music_media
['compact disc', 45, 'long playing record']
>>>
>>> music_media.insert(2, '8-track tape')
>>> music_media
['compact disc', 45, '8-track tape', 'long playing record']
```

In the preceding example, we initiated a list with a single element, then checked the list as we either inserted elements within the list, or appended new items at the end. Let's now determine if elements are in a list and how to find out the location of where items are in a list. We do this by using the **`in`** operator and `index()` method.

```
>>> 'cassette' in music_media
False
>>> 'compact disc' in music_media
True
>>> music_media.index(45)
1
>>> music_media.index('8-track tape')
2
>>> music_media.index('cassette')
Traceback (innermost last):
 File "<interactive input>", line 0, in ?
ValueError: list.index(x): x not in list
```

Oops! What happened in that last example? Well, it looks like using `index()` to check if items are in a list is not a good idea, because we get an error. It would be safer to check using the membership operator

in (or **not in**) first, and then using `index()` to find the element's location. We can put the last few calls to `index()` in a single **for** loop like this:

```
for eachMediaType in (45, '8-track tape', 'cassette'):
   if eachMediaType in music_media:
       print music_media.index(eachMediaType)
```

This solution helps us avoid the error we encountered above because `index()` is not called unless the object was found in the list. We will find out later how we can take charge if the error occurs, instead of bombing out as we did above.

We will now test drive `sort()` and `reverse()`, methods that will sort and reverse the elements of a list, respectively.

```
>>> music_media
['compact disc', 45, '8-track tape', 'long playing record']
>>> music_media.sort()
>>> music_media
[45, '8-track tape', 'compact disc', 'long playing record']
>>> music_media.reverse()
>>> music_media
['long playing record', 'compact disc', '8-track tape', 45]
```

Core Note: Mutable object methods that alter the object have no return value!

One very obvious place where new Python programmers get caught is when using methods that you think *should return a value. The most obvious one is* `sort()`*:*

```
>>> music_media.sort()        # where is the output?!?
>>>
```

The caveat about mutable object methods like `sort()`*,* `extend()`*, and* `reverse()` *is that these will perform their operation on a list in place, meaning that the contents of the existing list will be changed, but return* `None`*! Yes, it does fly in the face of string methods that* do *return values:*

```
>>> 'leanna, silly girl!'.upper()
'LEANNA, SILLY GIRL!'
```

Recall that strings are immutablemethods of immutable objects cannot modify them, so they do have to return a new object. If returning an object is a necessity for you, then we recommend that you look at the `reversed()` *and* `sorted()` *built-in functions introduced in Python 2.4.*

These work just like the list methods only they can be used in expressions because they do return objects. However, obviously the

original list object is left as is, and you are getting a new object back.

Going back to the `sort()` method, the default sorting algorithm employed by the `sort()` method is a derivative of MergeSort (modestly named "timsort"), which is O(lg(n!)). We defer all other explanation to the build files where you can get all the detailssource code: `Objects/listobject.c` and algorithm description: `Objects/listsort.txt`.

The `extend()` method will take the contents of one list and append its elements to another list:

```
>>> new_media = ['24/96 digital audio disc', 'DVD Audio
disc', 'Super Audio CD']
>>> music_media.extend(new_media)
>>> music_media
['long playing record', 'compact disc', '8-track tape',
45, '24/96 digital audio disc', 'DVD Audio disc', 'Super
Audio CD']
```

The argument to `extend()` can be any iterable, starting with 2.2. Prior to that, it had to be a sequence object, and prior to 1.6, it had to be a list. With an iterable (instead of a sequence), you can do more interesting things like:

```
>>> motd = []
>>> motd.append('MSG OF THE DAY')
>>> f = open('/etc/motd', 'r')
>>> motd.extend(f)
>>> f.close()
>>> motd
['MSG OF THE DAY', 'Welcome to Darwin!\n']
```

`pop()`, introduced in 1.5.2, will either return the last or requested item from a list and return it to the caller. We will see the `pop()` method in Section 6.15.1 as well as in the Exercises.

6.15. Special Features of Lists

6.15.1. Creating Other Data Structures Using Lists

Because of their container and mutable features, lists are fairly flexible and it is not very difficult to build other kinds of data structures using lists. Two that we can come up with rather quickly are stacks and queues.

Stack

A stack is a last-in-first-out (LIFO) data structure that works similarly to a cafeteria dining plate spring-loading mechanism. Consider the plates as objects. The first object off the stack is the last one you put in. Every new object gets "stacked" on top of the newest objects. To "push" an item on a stack is the terminology used to mean you are adding onto a stack. Likewise, to remove an element, you "pop" it off the stack. Example 6.3 shows a menu-driven program that implements a simple stack used to store strings.

Example 6.3. Using Lists as a Stack (`stack.py`)

This simple script uses lists as a stack to store and retrieve strings entered through this menu-driven text application using only the `append()` and `pop()` list methods.

```
#!/usr/bin/env python

stack = []

def pushit():
    stack.append(raw_input('Enter new string: ').strip())

def popit():
    if len(stack) == 0:
        print 'Cannot pop from an empty stack!'
    else:
        print 'Removed [', 'stack.pop()', ']'

def viewstack():
    print stack      # calls str() internally

CMDs = {'u': pushit, 'o': popit, 'v': viewstack}

def showmenu():
    pr = """
p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: """

```

```
28     while True:
29         while True:
30             try:
31                 choice = raw_input(pr).strip()[0].lower()
32             except (EOFError,KeyboardInterrupt,IndexError):
33                 choice = 'q'
34 
35             print '\nYou picked: [%s]' % choice
36             if choice not in 'uovq':
37                 print 'Invalid option, try again'
38             else:
39                 break
40 
41         if choice == 'q':
42             break
43         CMDs[choice]()
44 
45 if __name__ == '__main__':
46     showmenu()
```

Line-by-Line Explanation

Lines 13

In addition to the Unix startup line, we take this opportunity to clear the stack (a list).

Lines 56

The `pushit()` function adds an element (a string prompted from the user) to the stack.

Lines 812

The `popit()` function removes an element from the stack (the more recent one). An error occurs when trying to remove an element from an empty stack. In this case, a warning is sent back to the user. When an object is popped from the stack, the user sees which element was removed. We use single backquotes or backticks (` ) to symbolize the `repr()` command, showing the string complete with quotes, not just the contents of the string.

Lines 1415

The `viewstack()` function displays the current contents of the stack.

Line 17

Although we cover dictionaries formally in the next chapter, we wanted to give you a really small example of one here, a command vector (`CMDs`). The contents of the dictionary are the three "action" functions defined above, and they are accessed through the letter that the user must type to execute that command. For example, to push a string onto the stack, the user must enter `'u'`, so `'u'` is how access the `pushit()` from the dictionary. The chosen function is then executed on line 43.

Lines 1943

The entire menu-driven application is controlled from the `showmenu()` function. Here, the user is prompted with the menu options. Once the user makes a valid choice, the proper function is called. We have not covered exceptions and **`try-except`** statement in detail yet, but basically that section of the code allows a user to type ^D (EOF, which generates an `EOFError`) or ^C (interrupt to quit, which generates a `KeyboardInterrupt` error), both of which will be processed by our script in the same manner as if the user had typed the `'q'` to quit the application. This is one place where the exception-handling feature of Python comes in extremely handy. The outer **`while`** loop lets the user continue to execute commands until they quit the application while the inner one prompts the user until they enter a valid command option.

Lines 4546

This part of the code starts up the program if invoked directly. If this script were imported as a module, only the functions and variables would have been defined, but the menu would not show up. For more information regarding line 45 and the `__name__` variable, see Section 3.4.1.

Here is a sample execution of our script:

```
$ stack.py

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: u

You picked: [u]
Enter new string: Python

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: u

You picked: [u]
Enter new string: is

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: u

You picked: [u]
Enter new string: cool!

p(U)sh
p(O)p
(V)iew
```

```
(Q)uit

Enter choice: v

You picked: [v]
['Python', 'is', 'cool!']

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: o

You picked: [o]
Removed [ 'cool!' ]

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: o

You picked: [o]
Removed [ 'is' ]

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: o

You picked: [o]
Removed [ 'Python' ]

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: o

You picked: [o]
Cannot pop from an empty stack!

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: ^D

You picked: [q]
```

Queue

A queue is a first-in-first-out (FIFO) data structure, which works like a single-file supermarket or bank teller line. The first person in line is the first one served (and hopefully the first one to exit). New elements join by being "enqueued" at the end of the line, and elements are removed from the front by being "dequeued." The following code shows how, with a little modification from our stack script, we can implement a simple queue using lists.

Example 6.4. Using Lists as a Queue (`queue.py`)

This simple script uses lists as a queue to store and retrieve strings entered through this menu-driven text application, using only the `append()` and `pop()` list methods.

```
#!/usr/bin/env python

queue = []

def enQ():
    queue.append(raw_input('Enter new string: ').strip())

def deQ():
    if len(queue) == 0:
         print 'Cannot pop from an empty queue!'
    else:
        print 'Removed [', 'queue.pop(0)', ']'

def viewQ():
    print queue       # calls str() internally

CMDs = {'e': enQ, 'd': deQ, 'v': viewQ}

def showmenu():
    pr = """
(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: """

    while True:
        while True:
             try:
                 choice = raw_input(pr).strip()[0].lower()
             except (EOFError,KeyboardInterrupt,IndexError):
                 choice = 'q'

            print '\nYou picked: [%s]' % choice
            if choice not in 'devq':
               print 'Invalid option, try again'
            else:
                break

       if choice == 'q':
           break
       CMDs[choice]()

if __name__ == '__main__':
    showmenu()
```

Line-by-Line Explanation

Because of the similarities of this script with the `stack.py` script, we will describe in detail only the lines which have changed significantly:

Lines 17

The usual setup plus some constants for the rest of the script to use.

Lines 56

The `enQ()` function works exactly like `pushit()`, only the name has been changed.

Lines 812

The key difference between the two scripts lies here. The `deQ()` function, rather than taking the most recent item as `popit()` did, takes the oldest item on the list, the first element.

Lines 17, 21-24, 36

The `options` have been changed, so we need to reflect that in the prompt string and our validator.

We present some output here as well:

```
$ queue.py
(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: e

You picked: [e]
Enter new queue element: Bring out

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: e

You picked: [e]
Enter new queue element: your dead!

(E)nqueue
(D)equeue
(V)iew
(Q)uit
```

```
Enter choice: v

You picked: [v]
['Bring out', 'your dead!']

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: d

You picked: [d]
Removed [ 'Bring out' ]

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: d

You picked: [d]
    Removed [ 'your dead!' ]

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: d

You picked: [d]
Cannot dequeue from empty queue!

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: ^D
You picked: [q]
```

6.16. Tuples

Tuples are another container type extremely similar in nature to lists. The only visible difference between tuples and lists is that tuples use parentheses and lists use square brackets. Functionally, there is a more significant difference, and that is the fact that tuples are immutable. Because of this, tuples can do something that lists cannot do . . . be a dictionary key. Tuples are also the default when dealing with a group of objects.

Our usual *modus operandi* is to present the operators and built-in functions for the more general objects, followed by those for sequences and conclude with those applicable only for tuples, but because tuples share so many characteristics with lists, we would be duplicating much of our description from the previous section. Rather than providing much repeated information, we will differentiate tuples from lists as they apply to each set of operators and functionality, then discuss immutability and other features unique to tuples.

How to Create and Assign Tuples

Creating and assigning tuples are practically identical to creating and assigning lists, with the exception of empty tuplesthese require a trailing comma (`,`) enclosed in the tuple delimiting parentheses (`( )`) to prevent them from being confused with the natural grouping operation of parentheses. Do not forget the factory function!

```
>>> aTuple = (123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
>>> anotherTuple = (None, 'something to see here')
>>> print aTuple
(123, 'abc', 4.56, ['inner', 'tuple'], (7-9j))
>>> print anotherTuple
(None, 'something to see here')
>>> emptiestPossibleTuple = (None,)
>>> print emptiestPossibleTuple
(None,)
>>> tuple('bar')
('b', 'a', 'r')
```

How to Access Values in Tuples

Slicing works similarly to lists. Use the square bracket slice operator (`[ ]`) along with the index or indices.

```
>>> aTuple[1:4]
('abc', 4.56, ['inner', 'tuple'])

>>> aTuple[:3]
(123, 'abc', 4.56)
>>> aTuple[3][1]
'tuple'
```

How to Update Tuples

Like numbers and strings, tuples are immutable, which means you cannot update them or change values of tuple elements. In Sections 6.2 and 6.3.2, we were able to take portions of an existing string to create a new string. The same applies for tuples.

```
>>> aTuple = aTuple[0], aTuple[1], aTuple[-1]
>>> aTuple
(123, 'abc', (7-9j))
>>> tup1 = (12, 34.56)
>>> tup2 = ('abc', 'xyz')
>>> tup3 = tup1 + tup2
>>> tup3
(12, 34.56, 'abc', 'xyz')
```

How to Remove Tuple Elements and Tuples

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement to reduce an object's reference count. It will be deallocated when that count is zero. Keep in mind that most of the time one will just let an object go out of scope rather than using **del**, a rare occurrence in everyday Python programming.

```
del aTuple
```

6.17. Tuple Operators and Built-in Functions

6.17.1. Standard and Sequence Type Operators and Built-in Functions

Object and sequence operators and built-in functions act the exact same way toward tuples as they do with lists. You can still take slices of tuples, concatenate and make multiple copies of tuples, validate membership, and compare tuples.

Creation, Repetition, Concatenation

```
>>> t = (['xyz', 123], 23, -103.4)
>>> t
(['xyz', 123], 23, -103.4)
>>> t * 2
(['xyz', 123], 23, -103.4, ['xyz', 123], 23, -103.4)
>>> t = t + ('free', 'easy')
>>> t
(['xyz', 123], 23, -103.4, 'free', 'easy')
```

Membership, Slicing

```
>>> 23 in t
True
>>> 123 in t
False
>>> t[0][1]
123
>>> t[1:]
(23, -103.4, 'free', 'easy')
```

Built-in Functions

```
>>> str(t)
(['xyz', 123], 23, -103.4, 'free', 'easy')
>>> len(t)
5
>>> max(t)
'free'
>>> min(t)
-103.4
>>> cmp(t, (['xyz', 123], 23, -103.4, 'free', 'easy'))
0
>>> list(t)
[['xyz', 123], 23, -103.4, 'free', 'easy']
```

Operators

```
>>> (4, 2) < (3, 5)
```

```
False
>>> (2, 4) < (3, -1)
True
>>> (2, 4) == (3, -1)
False
>>> (2, 4) == (2, 4)
True
```

6.17.2. Tuple Type Operators and Built-in Functions and Methods

Like lists, tuples have no operators or built-in functions for themselves. All of the list methods described in the previous section were related to a list object's mutability, i.e., sorting, replacing, appending, etc. Since tuples are immutable, those methods are rendered superfluous, thus unimplemented.

PREV NEXT

6.18. Special Features of Tuples

6.18.1. How Are Tuples Affected by Immutability?

Okay, we have been throwing around this word "immutable" in many parts of the text. Aside from its computer science definition and implications, what is the bottom line as far as applications are concerned? What are all the consequences of an immutable data type?

Of the three standard types that are immutablenumbers, strings, and tuplestuples are the most affected. A data type that is immutable simply means that once an object is defined, its value cannot be updated, unless, of course, a completely new object is allocated. The impact on numbers and strings is not as great since they are scalar types, and when the sole value they represent is changed, that is the intended effect, and access occurs as desired. The story is different with tuples, however.

Because tuples are a container type, it is often desired to change single or multiple elements of that container. Unfortunately, this is not possible. Slice operators cannot show up on the left-hand side of an assignment. Recall this is no different for strings, and that slice access is used for read access only.

Immutability does not necessarily mean bad news. One bright spot is that if we pass in data to an API with which we are not familiar, we can be certain that our data will not be changed by the function called. Also, if we receive a tuple as a return argument from a function that we would like to manipulate, we can use the `list()` built-in function to turn it into a mutable list.

6.18.2. Tuples Are Not Quite So "Immutable"

Although tuples are defined as immutable, this does not take away from their flexibility. Tuples are not quite as immutable as we made them out to be. What do we mean by that? Tuples have certain behavioral characteristics that make them seem not as immutable as we had first advertised.

For example, we can join strings together to form a larger string. Similarly, there is nothing wrong with putting tuples together to form a larger tuple, so concatenation works. This process does not involve changing the smaller individual tuples in any way. All we are doing is joining their elements together. Some examples are presented here:

```
>>> s = 'first'
>>> s = s + ' second'
>>> s
'first second'
>>>
>>> t = ('third', 'fourth')
>>> t
('third', 'fourth')
>>>
>>> t = t + ('fifth', 'sixth')
>>> t
('third', 'fourth', 'fifth', 'sixth')
```

The same concept applies for repetition. Repetition is just concatenation of multiple copies of the same elements. In addition, we mentioned in the previous section that one can turn a tuple into a mutable list with a simple function call. Our final feature may surprise you the most. You can "modify" certain tuple elements. Whoa. What does that mean?

Although tuple objects themselves are immutable, this fact does not preclude tuples from containing mutable objects that *can* be changed.

```
>>> t = (['xyz', 123], 23, -103.4)
>>> t
(['xyz', 123], 23, -103.4)
>>> t[0][1]
123
>>> t[0][1] = ['abc', 'def']
>>> t
(['xyz', ['abc', 'def']], 23, -103.4)
```

In the above example, although `t` is a tuple, we managed to "change" it by replacing an item in the first tuple element (a list). We replaced `t[0][1]`, formerly an integer, with a list `['abc', 'def']`. Although we modified only a mutable object, in some ways, we also "modified" our tuple.

6.18.3. Default Collection Type

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., defaults to tuples, as indicated in these short examples:

```
>>> 'abc', -4.24e93, 18+6.6j, 'xyz'
('abc', -4.24e+093, (18+6.6j), 'xyz')
>>>
>>> x, y = 1, 2
>>> x, y
(1, 2)
```

Any function returning multiple objects (also no enclosing symbols) is a tuple. Note that enclosing symbols change a set of multiple objects returned to a single container object. For example:

```
def foo1():
    :
    return obj1, obj2, obj3
def foo2():
    :
    return [obj1, obj2, obj3]
def foo3():
    :
    return (obj1, obj2, obj3)
```

In the above examples, `foo1()` calls for the return of three objects, which come back as a tuple of three objects, `foo2()` returns a single object, a list containing three objects, and `foo3()` returns the same thing as `foo1()`. The only difference is that the tuple grouping is explicit.

Explicit grouping of parentheses for expressions or tuple creation is always recommended to avoid unpleasant side effects:

```
>>> 4, 2 < 3, 5     # int, comparison, int
```

```
(4, True, 5)
>>> (4, 2) < (3, 5) # tuple comparison
False
```

In the first example, the less than (<) operator took precedence over the comma delimiter intended for the tuples on each side of the less than sign. The result of the evaluation of `2 < 3` became the second element of a tuple. Properly enclosing the tuples enables the desired result.

6.18.4. Single-Element Tuples

Ever try to create a tuple with a single element? You tried it with lists, and it worked, but then you tried and tried with tuples, but you cannot seem to do it.

```
>>> ['abc']
['abc']

>>> type(['abc'])   # a list
<type 'list'>
>>>
>>> ('xyz')
'xyz'
>>> type(('xyz'))   # a string, not a tuple
<type 'str'>
```

It probably does not help your case that the parentheses are also overloaded as the expression grouping operator. Parentheses around a single element take on that binding role rather than serving as a delimiter for tuples. The workaround is to place a trailing comma `(,)` after the first element to indicate that this is a tuple and not a grouping.

```
>>> ('xyz',)
('xyz',)
```

6.18.5. Dictionary Keys

Immutable objects have values that cannot be changed. That means that they will always hash to the same value. That is the requirement for an object being a valid dictionary key. As we will find out in the next chapter, keys must be hashable objects, and tuples meet that criteria. Lists are not eligible.

Core Note: Lists versus Tuples

One of the questions in the Python FAQ asks, "Why are there separate tuple and list data types?" That question can also be rephrased as, "Do we really need two similar sequence types?" One reason why having lists and tuples is a good thing occurs in situations where having one is more advantageous than having the other.

One case in favor of an immutable data type is if you were manipulating sensitive data and were passing a mutable object to an unknown function (perhaps an API that you didn't even write!). As the engineer developing your piece of the software, you would definitely feel a lot more secure if you knew that the function you were calling could not alter the data.

An argument for a mutable data type is where you are managing dynamic data sets. You need to be able to create them on the fly, slowly or arbitrarily adding to them, or from time to time, deleting individual elements. This is definitely a case where the data type must be mutable. The good news is that with the `list()` *and* `tuple()` *built-in conversion functions, you can convert from one type to the other relatively painlessly.*

`list()` *and* `tuple()` *are functions that allow you to create a tuple from a list and vice versa. When you have a tuple and want a list because you need to update its objects, the* `list()` *function suddenly becomes your best buddy. When you have a list and want to pass it into a function, perhaps an API, and you do not want anyone to mess with the data, the* `tuple()` *function comes in quite useful.*

6.19. Related Modules

Table 6.12 lists the key related modules for sequence types. This list includes the `array` module to which we briefly alluded earlier. These are similar to lists except for the restriction that all elements must be of the same type. The `copy` module (see optional Section 6.20 below) performs shallow and deep copies of objects. The `operator` module, in addition to the functional equivalents to numeric operators, also contains the same four sequence types. The `types` module is a reference of type objects representing all types that Python supports, including sequence types. Finally, the `UserList` module contains a full class implementation of a list object. Because Python types cannot be subclassed, this module allows users to obtain a class that is list-like in nature, and to derive new classes or functionality. If you are unfamiliar with object-oriented programming, we highly recommend reading Chapter 13.

Table 6.12. Related Modules for Sequence Types

Module	*Contents*
`array`	Features the `array` restricted mutable sequence type, which requires all of its elements to be of the same type
`copy`	Provides functionality to perform shallow and deep copies of objects (see 6.20 below for more information)
`operator`	Contains sequence operators available as function calls, e.g., `operator.concat(`*`m, n`*`)` is equivalent to the concatenation (*`m + n`*) for sequences *`m`* and *`n`*
`re`	Perl-style regular expression search (and match); see Chapter 15
`StringIO/cStringIO`	Treats long strings just like a file object, i.e., `read()`, `seek()`, etc.; C-compiled version is faster but cannot be subclassed
`textwrap`[a]	Utility functions for wrapping/filling text fields; also has a class
`types`	Contains type objects for all supported Python types
`collections`[b]	High-performance container data types

[a] New in Python 2.3.

[b] New in Python 2.4.

6.20. *Copying Python Objects and Shallow and Deep Copies

Earlier in Section 3.5, we described how object assignments are simply object references. This means that when you create an object, then assign that object to another variable, Python does not copy the object. Instead, it copies only a *reference* to the object.

For example, let us say that you want to create a generic profile for a young couple; call it `person`. Then you copy this object for both of them. In the example below, we show two ways of copying an object, one uses slices and the other a factory function. To show we have three unrelated objects, we use the `id()` built-in function to show you each object's identity. (We can also use the **`is`** operator to do the same thing.)

```
>>> person = ['name', ['savings', 100.00]]
>>> hubby = person[:]       # slice copy
>>> wifey = list(person)    # fac func copy
>>> [id(x) for x in person, hubby, wifey]
[11826320, 12223552, 11850936]
```

Individual savings accounts are created for them with initial $100 deposits. The names are changed to customize each person's object. But when the husband withdraws $50.00, his actions affected his wife's account even though separate copies were made. (Of course, this is assuming that we want them to have separate accounts and not a single, joint account.) Why is that?

```
>>> hubby[0] = 'joe'
>>> wifey[0] = 'jane'
>>> hubby, wifey
(['joe', ['savings', 100.0]], ['jane', ['savings', 100.0]])
>>> hubby[1][1] = 50.00
>>> hubby, wifey
(['joe', ['savings', 50.0]], ['jane', ['savings', 50.0]])
```

The reason is that we have only made a *shallow copy*. A shallow copy of an object is defined to be a newly created object of the same type as the original object whose contents are references to the elements in the original object. In other words, the copied object itself is new, but the contents are not. Shallow copies of sequence objects are the default type of copy and can be made in any number of ways: (1) taking a complete slice `[:]`, (2) using a factory function, e.g., `list()`, `dict()`, etc., or (3) using the `copy()` function of the `copy` module.

Your next question should be: When the wife's name is assigned, how come it did not affect the husband's name? Shouldn't they both have the name `'jane'` now? The reason why it worked and we don't have duplicate names is because of the two objects in each of their lists, the first is immutable (a string) and the second is mutable (a list). Because of this, when shallow copies are made, the string is explicitly copied and a new (string) object created while the list only has its reference copied, not its members. So changing the names is not an issue but altering any part of their banking information is. Here, let us take a look at the object IDs for the elements of each list. Note that the banking object is exactly the same and the reason why changes to one affects the other. Note how, after we change their names, that the new name strings replace the original `'name'` string:

BEFORE:

```
>>> [id(x) for x in hubby]
[9919616, 11826320]
>>> [id(x) for x in wifey]
[9919616, 11826320]
```

AFTER:

```
>>> [id(x) for x in hubby]
[12092832, 11826320]
>>> [id(x) for x in wifey]
[12191712, 11826320]
```

If the intention was to create a joint account for the couple, then we have a great solution, but if we want separate accounts, we need to change something. In order to obtain a full or *deep copy* of the objectcreating a new container but containing references to completely new copies (references) of the element in the original objectwe need to use the `copy.deepcopy()` function. Let us redo the entire example but using deep copies instead:

```
>>> person = ['name', ['savings', 100.00]]
>>> hubby = person

>>> import copy
>>> wifey = copy.deepcopy(person)
>>> [id(x) for x in person, hubby, wifey]
[12242056, 12242056, 12224232]
>>> hubby[0] = 'joe'
>>> wifey[0] = 'jane'
>>> hubby, wifey
(['joe', ['savings', 100.0]], ['jane', ['savings', 100.0]])
>>> hubby[1][1] = 50.00
>>> hubby, wifey
(['joe', ['savings', 50.0]], ['jane', ['savings', 100.0]])
```

Now it is just the way we want it. For kickers, let us confirm that all four objects are different:

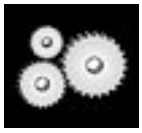
```
>>> [id(x) for x in hubby]
[12191712, 11826280]
>>> [id(x) for x in wifey]
[12114080, 12224792]
```

There are a few more caveats to object copying. The first is that non-container types (i.e., numbers, strings, and other "atomic" objects like `code`, `type`, and `xrange` objects) are not copied. Shallow copies of sequences are all done using complete slices. Finally, deep copies of tuples are not made if they contain only atomic objects. If we changed the banking information to a tuple, we would get only a shallow copy even though we asked for a deep copy:

```
>>> person = ['name', ('savings', 100.00)]
>>> newPerson = copy.deepcopy(person)
>>> [id(x) for x in person, newPerson]
[12225352, 12226112]
>>> [id(x) for x in person]
```

```
[9919616, 11800088]
>>> [id(x) for x in newPerson]
[9919616, 11800088]
```

Core Module: copy



The shallow and deep copy operations that we just described are found in the `copy` *module. There are really only two functions to use from this module:* `copy()` *creates shallow copy, and* `deepcopy()` *creates a deep copy.*

PREV NEXT

6.21. Summary of Sequences

Sequence types provide various mechanisms for ordered storage of data. Strings are a general medium for carrying data, whether it be displayed to a user, stored on a disk, transmitted across the network, or be a singular container for multiple sources of information. Lists and tuples provide container storage that allows for simple manipulation and access of multiple objects, whether they be Python data types or user-defined objects. Individual or groups of elements may be accessed as slices via sequentially ordered index offsets. Together, these data types provide flexible and easy-to-use storage tools in your Python development environment. We conclude this chapter with a summary of operators, built-in functions and methods for sequence types given in Table 6.13.

Table 6.13. Sequence Type Operators, Built-in Functions and Methods

Operator, Built-in Function or Method	*String*	*List*	*Tuple*
[] (list creation)		•	
()			•
""	•		
append()		•	
capitalize()	•		
center()	•		
chr()	•		
cmp()	•	•	•
count()	•	•	
decode()	•		
encode()	•		
endswith()	•		
expandtabs()	•		
extend()		•	
find()	•		
hex()	•		
index()	•	•	
insert()		•	

isdecimal()	•		
isdigit()	•		
islower()	•		
isnumeric()	•		
isspace()	•		
istitle()	•		
isupper()	•		
join()	•		
len()	•	•	•
list()	•	•	•
ljust()	•		
lower()	•		
lstrip()	•		
max()	•	•	•
min()	•	•	•
oct()	•		
ord()	•		
pop()		•	
raw_input()	•		
remove()		•	
replace()	•		
repr()	•	•	•
reverse()		•	
rfind()	•		
rindex()	•		
rjust()	•		
rstrip()	•		
sort()		•	
split()	•		
splitlines()	•		
startswith()	•		

`str()`	•	•	•
`strip()`	•		
`swapcase()`	•		
`split()`	•		
`title()`	•		
`tuple()`	•	•	•
`type()`	•	•	•
`upper()`	•		
`zfill()`	•		
`.`(attributes)	•	•	
`[]` (slice)	•	•	•
`[:]`	•	•	•
`*`	•	•	•
`%`	•		
`+`	•	•	•
`in`	•	•	•
`not in`	•	•	•

6.22. Exercises

6-1. *Strings*. Are there any string methods or functions in the string module that will help me determine if a string is part of a larger string?

6-2. *String Identifiers*. Modify the `idcheck.py` script in Example 6-1 such that it will determine the validity of identifiers of length 1 as well as be able to detect if an identifier is a keyword. For the latter part of the exercise, you may use the `keyword` module (specifically the `keyword.kwlist` list) to aid in your cause.

6-3. *Sorting.*

a.

Enter a list of numbers and sort the values in largest-to-smallest order.

b.

Do the same thing, but for strings and in reverse alphabetical (largest-to-smallest lexicographic) order.

6-4. *Arithmetic*. Update your solution to the test score exercise in the previous chapter such that the test scores are entered into a list. Your code should also be able to come up with an average score. See Exercises 2-9 and 5-3.

6-5. *Strings.*

a.

Update your solution to Exercise 2-7 so that you display a string one character at a time forward *and* backward.

b.

Determine if two strings match (without using comparison operators or the `cmp()` built-in function) by scanning each string. Extra credit: Add case-insensitivity to your solution.

c.

Determine if a string is palindromic (the same backward as it is forward). Extra credit: Add code to suppress symbols and whitespace if you want to process anything other than strict palindromes.

d.

Take a string and append a backward copy of that string, making a palindrome.

6-6. *Strings*. Create the equivalent to `string.strip()`: Take a string and remove all leading and trailing whitespace. (Use of `string.*strip()` defeats the purpose of this exercise.)

6-7. *Debugging*. Take a look at the code we present in Example 6.4 (`buggy.py`).

a.

Study the code and describe what this program does. Add a comment to every place you see a comment sign (#). Run the program.

b.

This problem has a big bug in it. It fails on inputs of 6, 12, 20, 30, etc., not to mention any even number in general. What is wrong with this program?

c.

Fix the bug in (b).

6-8. *Lists*. Given an integer value, return a string with the equivalent English text of each digit. For example, an input of 89 results in "eight-nine" being returned. Extra credit: Return English text with proper usage, i.e., "eighty-nine." For this part of the exercise, restrict values to be between 0 and 1,000.

6-9. *Conversion*. Create a sister function to your solution for Exercise 5.13 to take the total number of minutes and return the same time interval in hours and minutes, maximizing on the total number of hours.

6-10. *Strings*. Create a function that will return another string similar to the input string, but with its case inverted. For example, input of "Mr. Ed" will result in "mR. eD" as the output string.

Example 6.4. Buggy Program (`buggy.py`)

This is the program listing for Exercise 6-7. You will determine what this program does, add comments where you see "#"s, determine what is wrong with it, and provide a fix for it.

```
1  #!/usr/bin/env python
2
3  #
4  num_str = raw_input('Enter a number: ')
5
6  #
7  num_num = int(num_str)
8
9  #
10 fac_list = range(1, num_num+1)
11 print "BEFORE:", 'fac_list'
12
13 #
14 i = 0
15
16 #
17 while i < len(fac_list):
18
19     #
20     if num_num % fac_list[i] == 0:
21         del fac_list[i]
22     #
23     i = i + 1
25
26 #
27 print "AFTER:", 'fac_list'
```

6-11. *Conversion*.

a.

Create a program that will convert from an integer to an Internet Protocol (IP) address in the four-octet format of WWW.XXX.YYY.ZZZ.

b.

Update your program to be able to do the vice versa of the above.

6-12. *Strings*.

a.

Create a function called `findchr()`, with the following declaration:

```
def findchr(string, char)
```

`findchr()` will look for character char in *`string`* and return the index of the first occurrence of `char`, or `-1` if that *`char`* is not part of *`string`*. You cannot use `string.*find()` or `string.*index()` functions or methods.

b.

Create another function called `rfindchr()` that will find the last occurrence of a character in a string. Naturally this works similarly to `findchr()`, but it starts its search from the end of the input string.

c.

Create a third function called `subchr()` with the following declaration:

```
def subchr(string, origchar, newchar)
```

`subchr()` is similar to `findchr()` except that whenever *`origchar`* is found, it is replaced by *`newchar`*. The modified string is the return value.

6-13. *Strings*. The `string` module contains three functions, `atoi()`, `atol()`, and `atof()`, that convert strings to integers, long integers, and floating point numbers, respectively. As of Python 1.5, the Python built-in functions `int()`, `long()`, and `float()` can also perform the same tasks, in addition to `complex()`, which can turn a string into a complex number. (Prior to 1.5, however, those built-in functions converted only between numeric types.)

An `atoc()` was never implemented in the `string` module, so that is your task here. `atoc()` takes a single string as input, a string representation of a complex number, e.g., '`-1.23e+4-5.67j`', and returns the equivalent complex number object with the given value. You cannot use `eval()`, but `complex()` is available. However, you can only use `complex()` with the following restricted syntax: `complex(`*`real, imag`*`)` where `real` and `imag` are floating point values.

6-14. **Random Numbers*. Design a "rock, paper, scissors" game, sometimes called "Rochambeau," a game you may have played as a kid. Here are the rules. At the same time, using specified hand motions, both you and your opponent have to pick from one of the following: rock, paper, or scissors. The winner is determined by these rules, which form somewhat of a fun paradox:

a.

the paper covers the rock,

b.

the rock breaks the scissors,

c.

the scissors cut the paper. In your computerized version, the user enters his/her guess, the computer randomly chooses, and your program should indicate a winner or draw/tie. Note: The most algorithmic solutions use the fewest number of `if` statements.

6-15. *Conversion*.

a.

Given a pair of dates in some recognizable standard format such as MM/DD/YY or DD/MM/YY, determine the total number of days that fall between both dates.

b.

Given a person's birth date, determine the total number of days that person has been alive, including all leap days.

c.

Armed with the same information from (b) above, determine the number of days remaining until that person's next birthday.

6-16. *Matrices*. Process the addition and multiplication of a pair of M by N matrices.

6-17. *Methods*. Implement a function called `myPop()`, which is similar to the list `pop()` method. Take a list as input, remove the last object from the list and return it.

6-18. In the `zip()` example of Section 6.12.2, what does `zip(fn, ln)` return?

6-19. *Multi-Column Output*. Given any number of items in a sequence or other container, display them in equally-distributed number of columns. Let the caller provide the data and the output format. For example, if you pass in a list of 100 items destined for three columns, display the data in the requested format. In this case, two columns would have 33 items while the last would have 34. You can also let the user choose horizontal or vertical sorting.

PREV NEXT

Chapter 7. Mapping and Set Types

Chapter Topics

In this chapter, we take a look at Python's mapping and set types. As in earlier chapters, an introduction is followed by a discussion of the applicable operators and factory and built-in functions (BIFs) and methods. We then go into more specific usage of each data type.

7.1. Mapping Type: Dictionaries

Dictionaries are the sole mapping type in Python. Mapping objects have a one-to-many correspondence between *hashable* values (*keys*) and the objects they represent (*values*). They are similar to Perl hashes and can be generally considered as *mutable hash tables*. A dictionary object itself is mutable and is yet another container type that can store any number of Python objects, including other container types. What makes dictionaries different from sequence type containers like lists and tuples is the way the data are stored and accessed.

Sequence types use numeric keys only (numbered sequentially as indexed offsets from the beginning of the sequence). Mapping types may use most other object types as keys; strings are the most common. Unlike sequence type keys, mapping keys are often, if not directly, associated with the data value that is stored. But because we are no longer using "sequentially ordered" keys with mapping types, we are left with an unordered collection of data. As it turns out, this does not hinder our use because mapping types do not require a numeric value to index into a container to obtain the desired item. With a key, you are "mapped" directly to your value, hence the term "mapping type." The reason why they are commonly referred to as hash tables is because that is the exact type of object that dictionaries are. Dictionaries are one of Python's most powerful data types.

Core Note: What are hash tables and how do they relate to dictionaries?

Sequence types use sequentially ordered numeric keys as index offsets to store your data in an array format. The index number usually has nothing to do with the data value that is being stored. There should also be a way to store data based on another associated value such as a string. We do this all the time in everyday living. You file people's phone numbers in your address book based on last name, you add events to your calendar or appointment book based on date and time, etc. For each of these examples, an associated value to a data item was your key.

Hash tables are a data structure that does exactly what we described. They store each piece of data, called a value, based on an associated data item, called a key. Together, these are known as key-value pairs. *The hash table algorithm takes your key, performs an operation on it, called a hash function, and based on the result of the calculation, chooses where in the data structure to store your value. Where any one particular value is stored depends on what its key is. Because of this randomness, there is no ordering of the values in the hash table. You have an unordered collection of data.*

The only kind of ordering you can obtain is by taking either a dictionary's set of keys or values. The `keys()` *or* `values()` *method returns lists, which are sortable. You can also call* `items()` *to get a list of keys and values as tuple pairs and sort that. Dictionaries themselves have no implicit ordering because they are hashes.*

Hash tables generally provide good performance because lookups occur fairly quickly once you have a key.

Python dictionaries are implemented as resizeable hash tables. If you are familiar with Perl, then we can say that dictionaries are similar to Perl's associative arrays or hashes.

We will now take a closer look at Python dictionaries. The syntax of a dictionary entry is *`key:value`* Also, dictionary entries are enclosed in braces `( { } )`.

How to Create and Assign Dictionaries

Creating dictionaries simply involves assigning a dictionary to a variable, regardless of whether the dictionary has elements or not:

```
>>> dict1 = {}
>>> dict2 = {'name': 'earth', 'port': 80}
>>> dict1, dict2
({}, {'port': 80, 'name': 'earth'})
```

In Python versions 2.2 and newer, dictionaries may also be created using the factory function `dict()`. We discuss more examples later when we take a closer look at `dict()`, but here's a sneak peek for now:

```
>>> fdict = dict((['x', 1], ['y', 2]))
>>> fdict
{'y': 2, 'x': 1}
```

In Python versions 2.3 and newer, dictionaries may also be created using a very convenient built-in method for creating a "default" dictionary whose elements all have the same value (defaulting to `None` if not given), `fromkeys()`:

```
>>> ddict = {}.fromkeys(('x', 'y'), -1)
>>> ddict
{'y': -1, 'x': -1}
>>>
>>> edict = {}.fromkeys(('foo', 'bar'))
>>> edict
{'foo': None, 'bar': None}
```

How to Access Values in Dictionaries

To traverse a dictionary (normally by key), you only need to cycle through its keys, like this:

```
>>> dict2 = {'name': 'earth', 'port': 80}
>>>
>>>> for key in dict2.keys():
...     print 'key=%s, value=%s' % (key, dict2[key])
...
key=name, value=earth
key=port, value=80
```

Beginning with Python 2.2, you no longer need to use the `keys()` method to extract a list of keys to loop over. Iterators were created to simplify accessing of sequence-like objects such as dictionaries and files. Using just the dictionary name itself will cause an iterator over that dictionary to be used in a **`for`** `loop:`

```
>>> dict2 = {'name': 'earth', 'port': 80}
>>>
>>>> for key in dict2:
...     print 'key=%s, value=%s' % (key, dict2[key])
...
key=name, value=earth
key=port, value=80
```

To access individual dictionary elements, you use the familiar square brackets along with the key to obtain its value:

```
>>> dict2['name']
'earth'
>>>
>>> print 'host %s is running on port %d' % \
...  (dict2['name'], dict2['port'])
host earth is running on port 80
```

Dictionary `dict1` defined above is empty while `dict2` has two data items. The keys in `dict2` are `'name'` and `'port'`, and their associated value items are `'earth'` and `80`, respectively. Access to the value is through the key, as you can see from the explicit access to the `'name'` key.

If we attempt to access a data item with a key that is not part of the dictionary, we get an error:

```
>>> dict2['server']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyError: server
```

In this example, we tried to access a value with the key `'server'` which, as you know from the code above, does not exist. The best way to check if a dictionary has a specific key is to use the dictionary's `has_key()` method, or better yet, the **`in`** or **`not in`** operators starting with version 2.2. The `has_key()` method will be obsoleted in future versions of Python, so it is best to just use **`in`** or **`not in`**.

2.2

We will introduce all of a dictionary's methods below. The Boolean `has_key()` and the **`in`** and **`not in`** operators are Boolean, returning `true` if a dictionary has that key and `False` otherwise. (In Python versions preceding Boolean constants [older than 2.3], the values returned are 1 and 0, respectively.)

```
>>> 'server' in dict2 # or dict2.has_key('server')
False
>>> 'name' in dict # or dict2.has_key('name')
True
>>> dict2['name']
'earth'
```

Here is another dictionary example mixing the use of numbers and strings as keys:

```
>>> dict3 = {}
>>> dict3[1] = 'abc'
>>> dict3['1'] = 3.14159
>>> dict3[3.2] = 'xyz'
>>> dict3
{3.2: 'xyz', 1: 'abc', '1': 3.14159}
```

Rather than adding each key-value pair individually, we could have also entered all the data for `dict3` at the same time:

```
dict3 = {3.2: 'xyz', 1: 'abc', '1': 3.14159}
```

Creating the dictionary with a set key-value pair can be accomplished if all the data items are known in advance (obviously). The goal of the examples using `dict3` is to illustrate the variety of keys that you can use. If we were to pose the question of whether a key for a particular value should be allowed to change, you would probably say, "No." Right?

Not allowing keys to change during execution makes sense if you think of it this way: Let us say that you created a dictionary element with a key and value. Somehow during execution of your program, the key changed, perhaps due to an altered variable. When you went to retrieve that data value again with the original key, you got a `KeyError` (since the key changed), and you had no idea how to obtain your value now because the key had somehow been altered. For this reason, keys must be hashable, so numbers and strings are fine, but lists and other dictionaries are not. (See Section 7.5.2 for why keys must be hashable.)

How to Update Dictionaries

You can update a dictionary by adding a new entry or element (i.e., a key-value pair), modifying an existing entry, or deleting an existing entry (see below for more details on removing an entry).

```
>>> dict2['name'] = 'venus' # update existing entry
>>> dict2['port'] = 6969    # update existing entry
```

```
>>> dict2['arch'] = 'sunos5' # add new entry
>>>
>>> print 'host %(name)s is running on port %(port)d' %
dict2
host venus is running on port 6969
```

If the key does exist, then its previous value will be overridden by its new value. The **print** statement above illustrates an alternative way of using the string format operator (%), specific to dictionaries. Using the dictionary argument, you can shorten the `print` request somewhat because naming of the dictionary occurs only once, as opposed to occurring for each element using a tuple argument.

You may also add the contents of an entire dictionary to another dictionary by using the `update()` built-in method. We will introduce this method in Section 7.4.

How to Remove Dictionary Elements and Dictionaries

Removing an entire dictionary is not a typical operation. Generally, you either remove individual dictionary elements or clear the entire contents of a dictionary. However, if you really want to "remove" an entire dictionary, use the **del** statement (introduced in Section 3.5.5). Here are some deletion examples for dictionaries and dictionary elements:

```
del dict2['name']       # remove entry with key 'name'
dict2.clear()           # remove all entries in dict1
del dict2               # delete entire dictionary
dict2.pop('name')       # remove & return entry w/key
```

Core Tip: Avoid using built-in object names as identifiers for variables!

For those of you who began traveling in the Python universe before version 2.3, you may have once used `dict` *as an identifier for a dictionary. However, because* `dict()` *is now a type and factory function, overriding it may cause you headaches and potential bugs. The interpreter will allow such overriding-hey, it thinks you seem smart and look like you know what you are doing! So be careful. Do* NOT *use variables named after built-in types like:* `dict`, `list`, `file`, `bool`, `str`, `input`, *or* `len`!

7.2. Mapping Type Operators

Dictionaries will work with all of the standard type operators but do not support operations such as concatenation and repetition. Those operations, although they make sense for sequence types, do not translate to mapping types. In the next two subsections, we introduce you to the operators you can use with dictionaries.

7.2.1. Standard Type Operators

The standard type operators were introduced in Chapter 4. Here are some basic examples using some of those operators:

```
>>> dict4 = {'abc': 123}
>>> dict5 = {'abc': 456}
>>> dict6 = {'abc': 123, 98.6: 37}
>>> dict7 = {'xyz': 123}
>>> dict4 < dict5
True
>>> (dict4 < dict6) and (dict4 < dict7)
True
>>> (dict5 < dict6) and (dict5 < dict7)
True
>>> dict6 < dict7
False
```

How are all these comparisons performed? Like lists and tuples, the process is a bit more complex than it is for numbers and strings. The algorithm is detailed in Section 7.3.1.

7.2.2. Mapping Type Operators

Dictionary Key-Lookup Operator ([ ])

The only operator specific to dictionaries is the key-lookup operator, which works very similarly to the single element slice operator for sequence types.

For sequence types, an index offset is the sole argument or subscript to access a single element of a sequence. For a dictionary, lookups are by key, so that is the argument rather than an index. The key-lookup operator is used for both assigning values to and retrieving values from a dictionary:

```
d[k] = v      # set value 'v' in dictionary with key 'k'
d[k]          # lookup value in dictionary with key 'k'
```

(Key) Membership (`in, not in`)

Beginning with Python 2.2, programmers can use the **in** and **not in** operators to check key membership instead of the `has_key()` method:

```
>>> 'name' in dict2
True
>>> 'phone' in dict2
False
```

PREV NEXT

7.3. Mapping Type Built-in and Factory Functions

7.3.1. Standard Type Functions [type(), str(), and cmp()]

The type() factory function, when applied to a dict, returns, as you might expect, the dict type, "<type 'dict'>". The str() factory function will produce a printable string representation of a dictionary. These are fairly straightforward.

In each of the last three chapters, we showed how the cmp() BIF worked with numbers, strings, lists, and tuples. So how about dictionaries? Comparisons of dictionaries are based on an algorithm that starts with sizes first, then keys, and finally values. However, using cmp() on dictionaries isn't usually very useful.

The next subsection goes into further detail about the algorithm used to compare dictionaries, but this is advanced reading, and definitely optional since comparing dictionaries is not very useful or very common.

*Dictionary Comparison Algorithm

In the following example, we create two dictionaries and compare them, then slowly modify the dictionaries to show how these changes affect their comparisons:

```
>>> dict1 = {}
>>> dict2 = {'host': 'earth', 'port': 80}
>>> cmp(dict1, dict2)
-1
>>> dict1['host'] = 'earth'
>>> cmp(dict1, dict2)
-1
```

In the first comparison, dict1 is deemed smaller because dict2 has more elements (2 items vs. 0 items). After adding one element to dict1, it is still smaller (2 vs. 1), even if the item added is also in dict2.

```
>>> dict1['port'] = 8080
>>> cmp(dict1, dict2)
1
>>> dict1['port'] = 80
>>> cmp(dict1, dict2)
0
```

After we add the second element to dict1, both dictionaries have the same size, so their keys are then compared. At this juncture, both sets of keys match, so comparison proceeds to checking their values. The values for the 'host' keys are the same, but when we get to the 'port' key, dict2 is deemed larger because its value is greater than that of dict1's 'port' key (8080 vs. 80). When resetting dict2's 'port' key to the same value as dict1's 'port' key, then both dictionaries form equals: They have the same

size, their keys match, and so do their values, hence the reason that `0` is returned by `cmp()`.

```
>>> dict1['prot'] = 'tcp'
>>> cmp(dict1, dict2)
1
>>> dict2['prot'] = 'udp'
>>> cmp(dict1, dict2)
-1
```

As soon as an element is added to one of the dictionaries, it immediately becomes the "larger one," as in this case with `dict1`. Adding another key-value pair to `dict2` can tip the scales again, as both dictionaries' sizes match and comparison progresses to checking keys and values.

```
>>> cdict = {'fruits':1}
>>> ddict = {'fruits':1}
>>> cmp(cdict, ddict)
0
>>> cdict['oranges'] = 0
>>> ddict['apples'] = 0
>>> cmp(cdict, ddict)
14
```

Our final example reminds as that `cmp()` may return values other than -1, 0, or 1. The algorithm pursues comparisons in the following order.

(1) Compare Dictionary Sizes

If the dictionary lengths are different, then for `cmp` *`(dict1, dict2)`*, `cmp()` will return a positive number if *`dict1`* is longer and a negative number if *`dict2`* is longer. In other words, the dictionary with more keys is greater, i.e.,

`len(dict1) > len(dict2)` $\Longrightarrow$ `dict1 > dict2`

(2) Compare Dictionary Keys

If both dictionaries are the same size, then their keys are compared; the order in which the keys are checked is the same order as returned by the `keys()` method. (It is important to note here that keys that are the same will map to the same locations in the hash table. This keeps key-checking consistent.) At the point where keys from both do not match, they are directly compared and `cmp()` will return a positive number if the first differing key for *`dict1`* is greater than the first differing key of *`dict2`*.

(3) Compare Dictionary Values

If both dictionary lengths are the same and the keys match exactly, the values for each key in both dictionaries are compared. Once the first key with non-matching values is found, those values are compared directly. Then `cmp()` will return a positive number if, using the same key, the value in *`dict1`* is greater than the value in *`dict2`*.

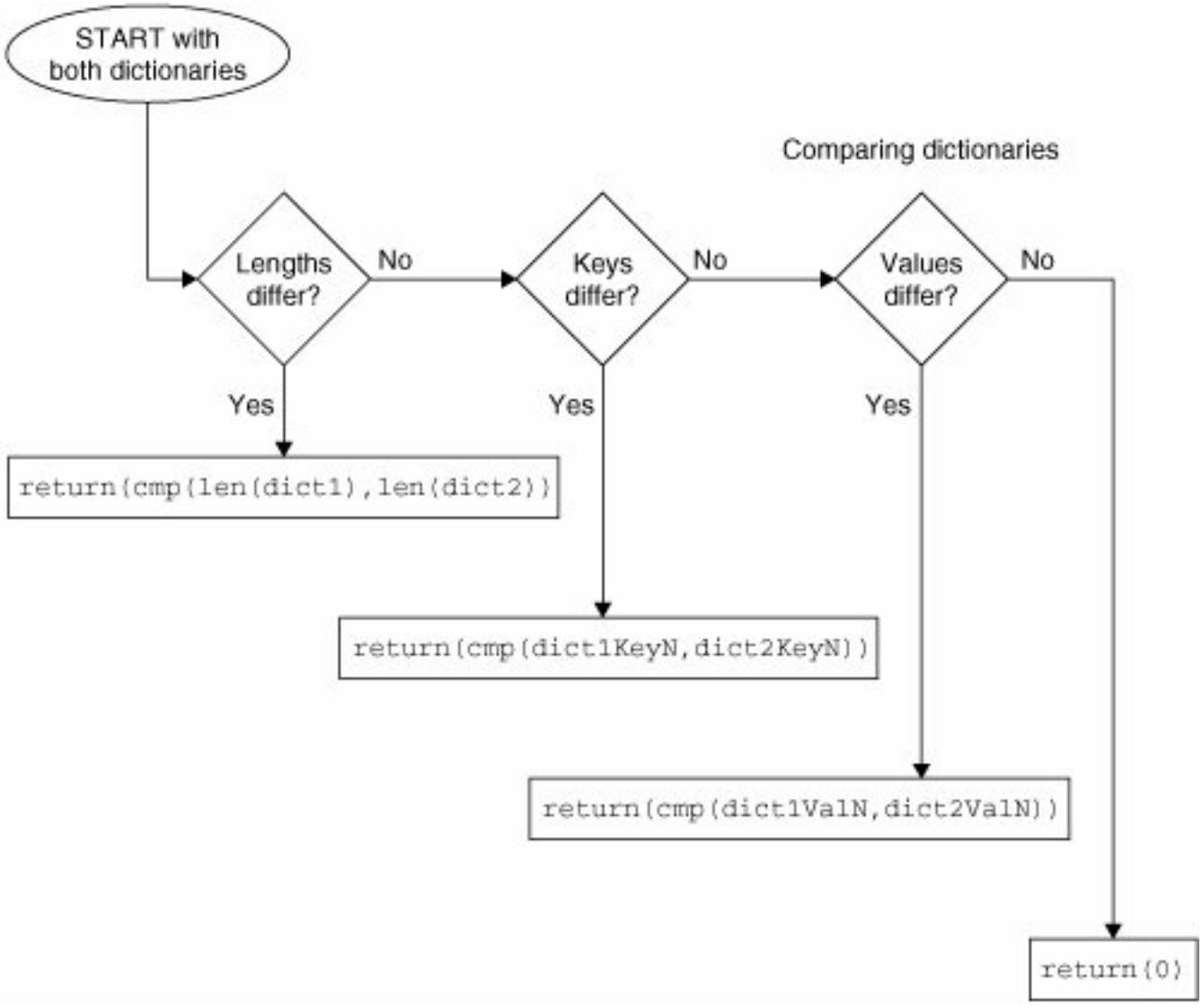
(4) Exact Match

If we have reached this point, i.e., the dictionaries have the same length, the same keys, and the same values for each key, then the dictionaries are an exact match and 0 is returned.

Figure 7-1 illustrates the dictionary compare algorithm we just outlined.

Figure 7-1. How dictionaries are compared

[View full size image]



7.3.2. Mapping Type Related Functions

dict ( )

The `dict()` factory function is used for creating dictionaries. If no argument is provided, then an empty dictionary is created. The fun happens when a container object is passed in as an argument to `dict()`.

If the argument is an iterable, i.e., a sequence, an iterator, or an object that supports iteration, then each element of the iterable must come in pairs. For each pair, the first element will be a new key in the dictionary with the second item as its value. Taking a cue from the official Python documentation for `dict ()`:

```
>>> dict(zip(('x', 'y'), (1, 2)))
{'y': 2, 'x': 1}
>>> dict([['x', 1], ['y', 2]])
{'y': 2, 'x': 1}
>>> dict([('xy'[i-1], i) for i in range(1,3)])
{'y': 2, 'x': 1}
```

If it is a(nother) mapping object, i.e., a dictionary, then `dict()` will just create a new dictionary and copy the contents of the existing one. The new dictionary is actually a shallow copy of the original one and the same results can be accomplished by using a dictionary's `copy()` built-in method. Because creating a new dictionary from an existing one using `dict()` is measurably slower than using `copy()`, we recommend using the latter.

Starting in Python 2.3, it is possible to call `dict()` with an existing dictionary or keyword argument dictionary (** function operator, covered in Chapter 11):

```
>>> dict(x=1, y=2)
{'y': 2, 'x': 1}
>>> dict8 = dict(x=1, y=2)
>>> dict8
{'y': 2, 'x': 1}
>>> dict9 = dict(**dict8)
>>> dict9
{'y': 2, 'x': 1}
```

We remind viewers that the `dict9` example is only an exercise in understanding the calling semantics of `dict()` and not a realistic example. It would be wiser (and better performance-wise) to execute something more along the lines of:

```
>>> dict9 = dict8.copy()
>>> dict9
{'y': 2, 'x' : 1}
```

len()

The `len()` BIF is flexible. It works with sequences, mapping types, and sets (as we will find out later on in this chapter). For a dictionary, it returns the total number of items, that is, key-value pairs:

```
>>> dict2 = {'name': 'earth', 'port': 80}
>>> dict2
{'port': 80, 'name': 'earth'}
>>> len(dict2)
2
```

We mentioned earlier that dictionary items are unordered. We can see that above, when referencing `dict2`, the items are listed in reverse order from which they were entered into the dictionary.

hash()

The `hash()` BIF is not really meant to be used for dictionaries per se, but it can be used to determine whether an object is fit to be a dictionary key (or not). Given an object as its argument, `hash()` returns the hash value of that object. The object can only be a dictionary key if it is *hashable* (meaning this function returns a[n integer] value without errors or raising an exception). Numeric values that are equal (when pitted against each other using a comparison operator) hash to the same value (even if their types differ). A `TypeError` will occur if an unhashable type is given as the argument to `hash()` (and consequently if an attempt is made to use such an object as the key when assigning a value to a dictionary):

```
>>> hash([])
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
>>>
>>> dict2[{}] = 'foo'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: dict objects are unhashable
```

In Table 7.1, we summarize these three mapping type related functions.

Table 7.1. Mapping Type Related Functions

Function	*Operation*
`dict([container])`	Factory function for creating a dictionary populated with items from *`container`*, if provided; if not, an empty dict is created
`len(mapping)`	Returns the length of *`mapping`* (number of key-value pairs)
`hash(obj)`	Returns hash value of *`obj`*

7.4. Mapping Type Built-in Methods

Dictionaries have an abundance of methods to help you get the job done, as indicated in Table 7.2.

Table 7.2. Dictionary Type Methods

Method Name	*Operation*
`dict.clear`[a] `()`	Removes all elements of *dict*
`dict.clear`[a] `()`	Returns a (shallow[b]) copy of *dict*
`dict.fromkeys`[c] `(seq, val=None)`	Creates and returns a new dictionary with the elements of *seq* as the keys and *val* as the initial value (defaults to `None` if not given) for all keys
`dict.get(key, default=None)`[a]	For key *key*, returns value or *default* if *key* not in `dict` (note that *default*'s default is `None`)
`dict.has_key (key)`	Returns `TRue` if `key` is in *dict*, `False` otherwise; partially deprecated by the **`in`** and **`not in`** operators in 2.2 but still provides a functional interface
`dict.items()`	Returns a list of the (key, value) tuple pairs of *dict*
`dict.keys()`	Returns a list of the keys of *dict*
`dict.iter *`[d] `()`	`iteritems(), iterkeys(), itervalues()` are all methods that behave the same as their non-iterator counterparts but return an iterator instead of a list
`dict.pop`[c] `(key [, default])`	Similar to `get()` but removes and returns `dict[key]` if `key` present and raises `KeyError` if `key` not in `dict` and `default` not given
`dict.setdefault (key, default=None)`[e]	Similar to `get()`, but sets `dict[key]=default` if `key` is not already in *dict*
`dict.update(dict2)`[a]	Add the key-value pairs of *dict2* to *dict*
`dict.values()`	Returns a list of the values of *dict*

[a] New in Python 1.5.

[b] More information regarding shallow and deep copies can be found in Section 6.19.

[c] New in Python 2.3.

[d] New in Python 2.2.

[e] New in Python 2.0.

Below, we showcase some of the more common dictionary methods. We have already seen `has_key()` and its replacements **`in`** and **`not in`** at work above. Attempting to access a nonexistent key will result in an exception (`KeyError`) as we saw in Section 7.1.

Basic dictionary methods focus on their keys and values. These are `keys()`, which returns a list of the dictionary's keys, `values()`, which returns a list of the dictionary's values, and `items()`, which returns a list of (key, value) tuple pairs. These are useful when you wish to iterate through a dictionary's keys or values, albeit in no particular order.

```
>>> dict2.keys()
['port', 'name']
>>>
>>> dict2.values()
[80, 'earth']
>>>
>>> dict2.items()
[('port', 80), ('name', 'earth')]
>>>
>>> for eachKey in dict2.keys():
... print 'dict2 key', eachKey, 'has value', dict2[eachKey]
...
dict2 key port has value 80
dict2 key name has value earth
```

The `keys()` method is fairly useful when used in conjunction with a *`for`* loop to retrieve a dictionary's values as it returns a list of a dictionary's keys. However, because its items (as with any keys of a hash table) are unordered, imposing some type of order is usually desired.

2.4

In Python versions prior to 2.4, you would have to call a dictionary's `keys()` method to get the list of its keys, then call that list's `sort()` method to get a sorted list to iterate over. Now a built-in function named `sorted()`, made especially for iterators, exists, which returns a sorted iterator:

```
>>> for eachKey in sorted(dict2):
...     print 'dict2 key', eachKey, 'has value',
dict2[eachKey]
...
dict2 key name has value earth
dict2 key port has value 80
```

The `update()` method can be used to add the contents of one directory to another. Any existing entries with duplicate keys will be overridden by the new incoming entries. Nonexistent ones will be added. All entries in a dictionary can be removed with the `clear()` method.

```
>>> dict2= {'host':'earth', 'port':80}
>>> dict3= {'host':'venus', 'server':'http'}
>>> dict2.update(dict3)
>>> dict2
{'server': 'http', 'port': 80, 'host': 'venus'}
>>> dict3.clear()
>>> dict3
{}
```

The `copy()` method simply returns a copy of a dictionary. Note that this is a shallow copy only. Again, see Section 6.19 regarding shallow and deep copies. Finally, the `get()` method is similar to using the key-lookup operator ([ ]), but allows you to provide a default value returned if a key does not exist. If a key does not exist and a default value is not given, then `None` is returned. This is a more flexible option than just using key-lookup because you do not have to worry about an exception being raised if a key does not exist.

```
>>> dict4 = dict2.copy()
>>> dict4
{'server': 'http', 'port': 80, 'host': 'venus'}
>>> dict4.get('host')
'venus'
>>> dict4.get('xxx')
>>> type(dict4.get('xxx'))
<type 'None'>
>>> dict4.get('xxx', 'no such key')
'no such key'
```

2.0

The built-in method, `setdefault()`, added in version 2.0, has the sole purpose of making code shorter by collapsing a common idiom: you want to check if a dictionary has a key. If it does, you want its value. If the dictionary does not have the key you are seeking, you want to set a default value and then return it. That is precisely what `setdefault()` does:

```
>>> myDict = {'host': 'earth', 'port': 80}
>>> myDict.keys()
['host', 'port']
>>> myDict.items()
[('host', 'earth'), ('port', 80)]
>>> myDict.setdefault('port', 8080)
80
>>> myDict.setdefault('prot', 'tcp')
'tcp'
>>> myDict.items()
[('prot', 'tcp'), ('host', 'earth'), ('port', 80)]
```

Earlier, we took a brief look at the `fromkeys()` method, but here are a few more examples:

```
>>> {}.fromkeys('xyz')
{'y': None, 'x': None, 'z': None}
>>>
>>> {}.fromkeys(('love', 'honor'), True)
{'love': True, 'honor': True}
```

Currently, the `keys(), items()`, and `values()` methods return lists. This can be unwieldy if such data collections are large, and the main reason why `iteritems(), iterkeys()`, and `itervalues()` were added to Python in 2.2. They function just like their list counterparts only they return iterators, which by lazier evaluation, are more memory-friendly. In future versions of Python, even more flexible and powerful objects will be returned, tentatively called *views*. Views are collection interfaces which give you access to container objects. For example, you may be able to delete a key from a view, which would then alter the corresponding dictionary accordingly.

PREV NEXT

7.5. Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, i.e., from standard objects to user-defined objects. However, the same cannot be said of keys.

7.5.1. More Than One Entry per Key Not Allowed

One rule is that you are constrained to having only one entry per key. In other words, multiple values per the same key are not allowed. (Container objects such as lists, tuples, and other dictionaries are fine.) When key *collisions* are detected (meaning duplicate keys encountered during assignment), the last (most recent) assignment wins.

```
>>> dict1 = {' foo':789, 'foo': 'xyz'}
>>> dict1
{'foo': 'xyz'}
>>>
>>> dict1['foo'] = 123
>>> dict1
{'foo': 123}
```

Rather than producing an error, Python does not check for key collisions because that would involve taking up memory for each key-value pair assigned. In the above example where the key `'foo'` is given twice on the same line, Python applies the key-value pairs from left to right. The value `789` may have been set at first, but is quickly replaced by the string `'xyz'`. When assigning a value to a nonexistent key, the key is created for the dictionary and value added, but if the key does exist (a collision), then its current value is replaced. In the above example, the value for the key `'foo'` is replaced twice; in the final assignment, `'xyz'` is replaced by `123`.

7.5.2. Keys Must Be Hashable

As we mentioned earlier in Section 7.1, most Python objects can serve as keys; however they have to be hashable objectsmutable types such as lists and dictionaries are disallowed because they cannot be hashed.

All immutable types are hashable, so they can definitely be used as keys. One caveat is numbers: Numbers of the same value represent the same key. In other words, the integer 1 and the float 1.0 hash to the same value, meaning that they are identical as keys.

Also, there are some mutable objects that are (barely) hashable, so they are eligible as keys, but there are very few of them. One example would be a class that has implemented the `__hash__()` special method. In the end, an immutable value is used anyway as `__hash__()` must return an integer.

Why must keys be hashable? The hash function used by the interpreter to calculate where to store your data is based on the value of your key. If the key was a mutable object, its value could be changed. If a key changes, the hash function will map to a different place to store the data. If that was the case, then the hash function could never reliably store or retrieve the associated value. Hashable keys were chosen for the very fact that their values cannot change. (This question can also be found in the Python FAQ.)

We know that numbers and strings are allowed as keys, but what about tuples? We know they are immutable, but in Section 6.17.2, we hinted that they might not be as immutable as they could be. The clearest example of that was when we modified a list object that was one of our tuple elements. To allow tuples as valid keys, one more restriction must be enacted: Tuples are valid keys only if they only contain immutable arguments like numbers and strings.

We conclude this chapter on dictionaries by presenting a program (`userpw.py` as in Example 7.1) that manages usernames and passwords in a mock login entry database system. This script accepts new users given that they provide a login name and a password. Once an "account" has been set up, an existing user can return as long as the user gives the login and correct password. New users cannot create an entry with an existing login name.

Example 7.1. Dictionary Example (`userpw.py`)

This application manages a set of users who join the system with a login name and a password. Once established, existing users can return as long as they remember their login and password. New users cannot create an entry with someone else's login name.

```
1  #!/usr/bin/env python
2
3  db = {}
4
5  def newuser():
6      prompt = 'login desired: '
7      while True:
8          name = raw_input(prompt)
9          if db.has_key(name):
10             prompt = 'name taken, try another: '
11             continue
12         else:
13             break
14     pwd = raw_input('passwd: ')
15     db[name] = pwd
16
17 def olduser():
18     name = raw_input('login: ')
19     pwd = raw_input('passwd: ')
20     passwd = db.get(name)
21     if passwd == pwd:
22         print 'welcome back', name
23     else:
24         print 'login incorrect'
25
26 def showmenu():
27     prompt = """
28 (N)ew User Login
29 (E)xisting User Login
30 (Q)uit
31
32 Enter choice: """
33
34 done = False
35     while not done:
36
37         chosen = False
38         while not chosen:
```

```
39             try:
40                 choice =
   raw_input(prompt).strip()[0].lower()
41             except (EOFError, KeyboardInterrupt):
42                 choice = 'q'
43             print '\nYou picked: [%s]' % choice
44             if choice not in 'neq':
45                 print 'invalid option, try again'
46             else:
47                 chosen = True
48
49         if choice == 'q': done = True
50         if choice == 'n': newuser()
51         if choice == 'e': olduser()
52
53 if __name__ == '__main__':
54     showmenu()
```

Line-by-Line Explanation

Lines 13

After the Unix-startup line, we initialize the program with an empty user database. Because we are not storing the data anywhere, a new user database is created every time this program is executed.

Lines 515

The `newuser()` function is the code that serves new users. It checks to see if a name has already been taken, and once a new name is verified, the user is prompted for his or her password (no encryption exists in our simple program), and his or her password is stored in the dictionary with his or her user name as the key.

Lines 1724

The `olduser()` function handles returning users. If a user returns with the correct login and password, a welcome message is issued. Otherwise, the user is notified of an invalid login and returned to the menu. We do not want an infinite loop here to prompt for the correct password because the user may have inadvertently entered the incorrect menu option.

Lines 2651

The real controller of this script is the `showmenu()` function. The user is presented with a friendly menu. The prompt string is given using triple quotes because it takes place over multiple lines and is easier to manage on multiple lines than on a single line with embedded `'\n'` symbols. Once the menu is displayed, it waits for valid input from the user and chooses which mode of operation to follow based on the menu choice. The **`try-except`** statements we describe here are the same as for the `stack.py` and `queue.py` examples from the last chapter (see Section 6.14.1).

Lines 5354

This is the familiar code that will only call `showmenu()` to start the application if the script was involved directly (not imported). Here is a sample execution of our script:

```
$ userpw.py

(N)ew User Login
(E)xisting User Login
(Q)uit

Enter choice: n

You picked: [n]
login desired: king arthur
passwd: grail

(N)ew User Login
(E)xisting User Login
(Q)uit

Enter choice: e

You picked: [e]
login: sir knight
passwd: flesh wound
login incorrect

(N)ew User Login
(E)xisting User Login
(Q)uit

Enter choice: e

You picked: [e]
login: king arthur
passwd: grail
welcome back king arthur

(N)ew User Login
(E)xisting User Login
(Q)uit

Enter choice: ^D
You picked: [q]
```

7.6. Set Types

In mathematics, a set is any collection of distinct items, and its members are often referred to as set elements. Python captures this essence in its set type objects. A set object is an unordered collection of hashable values. Yes, set members would make great dictionary keys. Mathematical sets translate to Python set objects quite effectively and testing for set membership and operations such as union and intersection work in Python as expected.

Like other container types, sets support membership testing via `in` and `not in` operators, cardinality using the `len()` BIF, and iteration over the set membership using for loops. However, since sets are unordered, you do not index into or slice them, and there are no keys used to access a value.

There are two different types of sets available, mutable (`set`) and immutable (`frozenset`). As you can imagine, you are allowed to add and remove elements from the mutable form but not the immutable. Note that mutable sets are not hashable and thus cannot be used as either a dictionary key or as an element of another set. The reverse is true for frozen sets, i.e., they have a hash value and can be used as a dictionary key or a member of a set.

Sets became available in Python 2.3 via the `sets` module and accessed via the `ImmutableSet` and `Set` classes. However, it was decided that having them as built-in types was a better idea, so these classes were then ported to C along with some improvements and integrated into Python 2.4. You can read more about those improvements as well as set types in general in PEP 218 at http://python.org/peps/pep-0218.html.

Although sets are now an official Python type, they have often been seen in many Python applications (as user-defined classes), a wheel that has been reinvented many times over, similar to complex numbers (which eventually became a Python type way back in 1.4). Until current versions of Python, most users have tried to shoehorn set functionality into standard Python types like lists and dictionaries as proxies to a real set type (even if they were not the perfect data structure for their applications). Now users have more options, including a "real" set type.

Before we go into detail regarding Python set objects, we have to mentally translate the mathematical symbols to Python (see Table 7.3) so that we are clear on terminology and functionality.

Table 7.3. Set Operation and Relation Symbols

Mathematical Symbol	*Python Symbol*	*Description*
∈	`in`	Is a member of
∉	`not in`	Is not a member of
=	==	Is equal to

≠	!=	Is not equal to
⊂	<	Is a (strict) subset of
⊆	<=	Is a subset of (includes improper subsets)
⊃	>	Is a (strict) superset of
⊇	>=	Is a superset of (includes improper supersets)
∩	&	Intersection
∪	|	Union
- or \	-	Difference or relative complement
Δ	^	Symmetric difference

How to Create and Assign Set Types

There is no special syntax for sets like there is for lists ([ ]) and dictionaries ({ }) for example. Lists and dictionaries can also be created with their corresponding factory functions `list()` and `dict()`, and that is also the only way sets can be created, using *their* factory functions `set()` and `frozenset()`:

```
>>> s = set('cheeseshop')
>>> s
set(['c', 'e', 'h', 'o', 'p', 's'])
>>> t = frozenset('bookshop')
>>> t
frozenset(['b', 'h', 'k', 'o', 'p', 's'])
>>> type(s)
<type 'set'>
>>> type(t)
<type 'frozenset'>
>>> len(s)
6
>>> len(s) == len(t)
True
>>> s == t
False
```

How to Access Values in Sets

You are either going to iterate through set members or check if an item is a member (or not) of a set:

```
>>> 'k' in s
False
>>> 'k' in t
True
>>> 'c' not in t
True

>>> for i in s:
```

```
... print i
...
c
e
h
o
p
s
```

How to Update Sets

You can add and remove members to and from a set using various built-in methods and operators:

```
>>> s.add('z')
>>> s
set(['c', 'e', 'h', 'o', 'p', 's', 'z'])
>>> s.update('pypi')
>>> s
set(['c', 'e', 'i', 'h', 'o', 'p', 's', 'y', 'z'])
>>> s.remove('z')
>>> s
set(['c', 'e', 'i', 'h', 'o', 'p', 's', 'y'])
>>> s -= set('pypi')
>>> s
set(['c', 'e', 'h', 'o', 's'])
```

As mentioned before, only mutable sets can be updated. Any attempt at such operations on immutable sets is met with an exception:

```
>>> t.add('z')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'frozenset' object has no attribute 'add'
```

How to Remove Set Members and Sets

We saw how to remove set members above. As far as removing sets themselves, like any Python object, you can let them go out of scope or explicitly remove them from the current namespace with **`del`**. If the reference count goes to zero, then it is tagged for garbage collection.

```
>>> del s
>>>
```

7.7. Set Type Operators

7.7.1. Standard Type Operators (all set types)

Membership (`in, not in`)

As for sequences, Python's in and not in operators are used to determine whether an element is (or is not) a member of a set.

```
>>> s = set('cheeseshop')
>>> t = frozenset('bookshop')
>>> 'k' in s
False
>>> 'k' in t
True
>>> 'c'  not in t
True
```

Set Equality/Inequality

Equality (or inequality) may be checked between the same or different set types. Two sets are equal if and only if every member of each set is a member of the other. You can also say that each set must be a (n improper) subset of the other, e.g., both expressions `s <= t` and `s >= t` are true, or (`s <= t and s >= t`) `is TRue`. Equality (or inequality) is independent of set type or ordering of members when the sets were createdit is all based on the set membership.

```
>>> s == t
False
>>> s != t
True
>>> u = frozenset(s)
>>> s == u
True
>>> set('posh') == set('shop')
True
```

Subset Of/Superset Of

Sets use the Python comparison operators to check whether sets are subsets or supersets of other sets. The "less than" symbols (`<, <=`) are used for subsets while the "greater than" symbols (`>, >=`) are used for supersets.

Less-than and greater-than imply strictness, meaning that the two sets being compared cannot be equal to each other. The equal sign allows for less strict improper subsets and supersets.

Sets support both proper (`<`) and improper (`<=`) subsets as well as proper (`>`) and improper (`>=`) supersets. A set is "less than" another set if and only if the first set is a proper subset of the second set (is a subset but not equal), and a set is "greater than" another set if and only if the first set is a proper

superset of the second set (is a superset but not equal).

```
>>> set('shop') < set('cheeseshop')
True
>>> set('bookshop') >= set('shop')
True
```

7.7.2. Set Type Operators (All Set Types)

Union (|)

The union operation is practically equivalent to the OR (or inclusive disjunction) of sets. The union of two sets is another set where each element is a member of at least one of the sets, i.e., a member of one set *or* the other. The union symbol has a method equivalent, `union()`.

```
>>> s | t
set(['c', 'b', 'e', 'h', 'k', 'o', 'p', 's'])
```

Intersection (&)

You can think of the intersection operation as the AND (or conjunction) of sets. The intersection of two sets is another set where each element must be a member of at both sets, i.e., a member of one set *and* the other. The intersection symbol has a method equivalent, `intersection()`.

```
>>> s & t
set(['h', 's', 'o', 'p']
```

Difference/Relative Complement (-)

The difference, or relative complement, between two sets is another set where each element is in one set but not the other. The difference symbol has a method equivalent, `difference()`.

```
>>> s - t
set(['c', 'e'])
```

Symmetric Difference (^)

Similar to the other Boolean set operations, symmetric difference is the XOR (or exclusive disjunction) of sets. The symmetric difference between two sets is another set where each element is a member of one set but not the other. The symmetric difference symbol has a method equivalent, `symmetric_difference()`.

```
>>> s ^ t
set(['k', 'b', 'e', 'c'])
```

Mixed Set Type Operations

In the above examples, `s` is a set while `t` is a frozenset. Note that each of the resulting sets from using the set operators above result in sets. However note that the resulting type is different when the operands are reversed:

```
>>> t | s
frozenset(['c', 'b', 'e', 'h', 'k', 'o', 'p', 's'])
>>> t ^ s
frozenset(['c', 'b', 'e', 'k'])
>>> t - s
frozenset(['k', 'b'])
```

If both types are sets or frozensets, then the type of the result is the same type as each of the operands, but if operations are performed on mixed types (set and frozenset, and vice versa), the type of the resulting set is the same type as the left operand, which we can verify in the above.

And no, the plus sign is not an operator for the set types:

```
>>> v = s + t
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'set' and
'set'
>>> v = s | t
>>> v
set(['c', 'b', 'e', 'h', 'k', 'o', 'p', 's'])
>>> len(v)
8
>>> s < v
True
```

7.7.3. Set Type Operators (Mutable Sets Only)

(Union) Update (`|=`)

The update operation adds (possibly multiple) members from another set to the existing set. The method equivalent is `update()`.

```
>>> s = set('cheeseshop')
>>> u = frozenset(s)
>>> s |= set('pypi')
>>> s
set(['c', 'e', 'i', 'h', 'o', 'p', 's', 'y'])
```

Retention/Intersection Update (`&=`)

The retention (or intersection update) operation keeps only the existing set members that are also elements of the other set. The method equivalent is `intersection_update()`.

```
>>> s = set(u)
```

```
>>> s &= set('shop')
>>> s
set(['h', 's', 'o', 'p'])
```

Difference Update (- =)

The difference update operation returns a set whose elements are members of the original set after removing elements that are (also) members of the other set. The method equivalent is `difference_update()`.

```
>>> s = set(u)
>>> s -= set('shop')
>>> s
set(['c', 'e'])
```

Symmetric Difference Update (^ =)

The symmetric difference update operation returns a set whose members are either elements of the original or other set but not both. The method equivalent is `symmetric_difference_update()`.

```
>>> s = set(u)
>>> t = frozenset('bookshop')
>>> s ^= t
>>> s
set(['c', 'b', 'e', 'k'])
```

7.8. Built-in Functions

7.8.1. Standard Type Functions

`len()`

The `len()` BIF for sets returns cardinality (or the number of elements) of the set passed in as the argument.

```
>>> s = set(u)
>>> s
set(['p', 'c', 'e', 'h', 's', 'o'])
>>> len(s)
6
```

7.8.2. Set Type Factory Functions

`set()` and `frozenset()`

The `set()` and `frozenset()` factory functions generate mutable and immutable sets, respectively. If no argument is provided, then an empty set is created. If one is provided, it must be an iterable, i.e., a sequence, an iterator, or an object that supports iteration such as a file or a dictionary.

```
>>> set()
set([])
>>> set([])
set([])
>>> set(())
set([])
>>> set('shop')
set(['h', 's', 'o', 'p'])
>>>
>>> frozenset(['foo', 'bar'])
frozenset(['foo', 'bar'])
>>>
>>> f = open('numbers', 'w')
>>> for i in range(5):
...     f.write('%d\n' % i)
...
>>> f.close()
>>> f = open('numbers', 'r')
>>> set(f)
set(['0\n', '3\n', '1\n', '4\n', '2\n'])
>>> f.close()
```

7.9. Set Type Built-in Methods

7.9.1. Methods (All Set Types)

We have seen the operator equivalents to most of the built-in methods, summarized in Table 7.4.

Table 7.4. Set Type Methods

Method Name	*Operation*
`s.issubset(t)`	Returns `TRue` if every member of *s* is in *t*, `False` otherwise
`s.issuperset(t)`	Returns `true` if every member of *s* is in *t*, `False` otherwise
`s.union(t)`	Returns a new set with the members of *s* or *t*
`s.intersection(t)`	Returns a new set with members of *s* and *t*
`s.difference(t)`	Returns a new set with members of *s* but not *t*
`s.symmetric_difference(t)`	Returns a new set with members of *s* or *t* but not both
`s.copy()`	Returns a new set that is a (shallow) copy of *s*

The one method without an operator equivalent is `copy()`. Like the dictionary method of the same name, it is faster to create a copy of the object using `copy()` than it is using a factory function like `set()`, `frozenset()`, or `dict()`.

7.9.2. Methods (Mutable Sets Only)

Table 7.5 summarizes all of the built-in methods that only apply to mutable sets, and similar to the methods above, we have already seen most of their operator equivalents.

Table 7.5. Mutable Set Type Methods

Method Name	*Operation*
`s.update(t)`	Updates `s` with elements added from *t*; in other words, `s` now has members of either *s* or *t*
`s.intersection_update(t)`	Updates *s* with members of both *s* and *t*
`s.difference_update(t)`	Updates *s* with members of *s* without elements of *t*

`s.symmetric_difference_update(t)`	Updates *s* with members of *s* or *t* but not both
s`.add(`*obj*`)`	Adds object *obj* to set *s*
s`.remove(`*obj*`)`	Removes object *obj* from set *s*; `KeyError` raised if *obj* is not an element of `s (obj not in s)`
s`.discard(`*obj*`)`	Removes object *obj* if *obj* is an element of *s* `(`*obj* `in` *s*`)`
s`.pop()`	Removes and returns an arbitrary object of *s*
s`.clear()`	Removes all elements from *s*

The new methods here are `add(), remove(), discard(), pop()`, and `clear()`. For the methods that take an object, the argument must be hashable.

7.9.3. Using Operators versus Built-in Methods

As you can see, there are many built-in methods that have near-equivalents when using operators. By "near-equivalent," we mean that there is one major difference: when using the operators, both operands must be sets while for the methods, objects can be iterables too. Why was it implemented this way? The official Python documentation states that "[this] precludes error-prone constructions like `set('abc')` [and] `'cbs'` in favor of the more readable `set('abc').intersection('cbs')`."

7.10. Operator, Function/Method Summary Table for Set Types

In Table 7.6, we summarize all of the set type operators, functions, and methods.

Table 7.6. Set Type Operators, Functions, and Methods

Function/Method Name	*Operator Equivalent*	*Description*
All Set Types		
`len(s)`		Set cardinality: number of elements in `s`
`set([obj])`		Mutable set factory function; if `obj` given, it must be iterable, new set elements taken from `obj`; if not, creates an empty set
`frozenset([obj])`		Immutable set factory function; operates the same as `set()` except returns immutable set
	`obj in s`	Membership test: is `obj` an element of `s`?
	`obj not in s`	Non-membership test: is `obj` not an element of`s`?
	`s == t`	Equality test: do `s` and `t` have exactly the same elements?
	`s != t`	Inequality test: opposite of ==
	`s < t`	(Strict) subset test; `s != t` and all elements of `s` are members of `t`
`s.issubset(t)`	`s <= t`	Subset test (allows improper subsets): all elements of `s` are members of `t`
	`s > t`	(Strict) superset test: `s != t` and all elements of `t` are members of `s`
`s.issuperset(t)`	`s >= t`	Superset test (allows improper supersets): all elements of `t` are members of `s`
`s.union(t)`	`s | t`	Union operation: elements in `s` or `t`
`s.intersection(t)`	`s & t`	Intersection operation: elements in `s` and `t`
`s.difference(t)`	`s - t`	Difference operation: elements in `s` that are not elements of `t`

s.symmetric_ difference(t)	*s* ^ *t*	Symmetric difference operation: elements of either *s* or *t* but not both
s.copy()		Copy operation: return (shallow) copy of *s*

Mutable Sets Only

s.update(t)	*s* |= *t*	(Union) update operation: members of *t* added to *s*
s.intersection_update(t)	*s* &= *t*	Intersection update operation: *s* only contains members of the original *s* and *t*
s.difference_ update(t)	*s* -= *t*	Difference update operation: s only contains original members who are not in t
s.symmetric_ difference_ update(t)	*s* ^= *t*	Symmetric difference update operation: *s* only contains members of *s* or *t* but not both
s.add(*obj*)		Add operation: add *obj* to s
s.remove(*obj*)		Remove operation: remove *obj* from *s*; KeyError raised if *obj* not in s
s.discard(*obj*)		Discard operation: friendlier version of remove()remove *obj* from *s* if *obj* in *s*
s.pop()		Pop operation: remove and return an arbitrary element of *s*
s.clear()		Clear operation: remove all elements of *s*

7.11. Related Modules

The `sets` module became available in 2.3 and may be useful if you wish to subclass the `Set` or `ImmutableSet` classes. Although set types were integrated into Python 2.4, there are currently no plans to deprecate the module.

Some general online references for sets which you may find useful include:

http://en.wikipedia.org/wiki/Set

http://www.geocities.com/basicmathsets/set.html

http://www.math.uah.edu/stat/foundations/Sets.xhtml

7.12. Exercises

7-1. *Dictionary Methods*. What dictionary method would we use to combine two dictionaries together?

7-2. *Dictionary Keys*. We know that dictionary values can be arbitrary Python objects, but what about the keys? Try using different types of objects as the key other than numbers or strings. What worked for you and what didn't? As for the failures, why do you think they didn't succeed?

7-3. *Dictionary and List Methods.*

a.

Create a dictionary and display its keys alphabetically.

b.

Now display both the keys and values sorted in alphabetical order by the key.

c.

Same as part (b), but sorted in alphabetical order by the value. (Note: This has no practical purpose in dictionaries or hash tables in general because most access and ordering [if any] is based on the keys. This is merely an exercise.)

7-4. *Creating Dictionaries*. Given a pair of identically sized lists, say, `[1, 2, 3,...]`, and `['abc', 'def', 'ghi',...]`, process all that list data into a single dictionary that looks like: `{ 1:'abc', 2:'def', 3:'ghi',...}`.

7-5. *userpw2.py*. The following problems deal with the program in Example 7.1, a manager of a database of name-password key-value pairs.

a.

Update the script so that a timestamp (see the `time` module) is also kept with the password indicating date and time of last login. This interface should prompt for login and password and indicate a successful or failed login as before, but if successful, it should update the last login timestamp. If the login occurs within four hours of the last login, tell the user, "You already logged in at: *<last_ login_timestamp>*."

b.

Add an "administration" menu to include the following two menu options: (1) remove a user and (2) display a list of all users in the system and their passwords

c.

The passwords are currently not encrypted. Add password-encryption if so desired (see the `crypt, rotor`, or other cryptographic modules).

d.

*Add a GUI interface, i.e., Tkinter, on top of this application.

e.

Allow usernames to be case-insensitive.

f.

Restrict usernames by not allowing symbols or whitespace.

g.

Merge the "new user" and "old user" options together. If a new user tries to log in with a nonexistent username, prompt if they are new and if so, do the proper setup. Otherwise, they are an existing user so log in as normal.

7-6. *Lists and Dictionaries*. Create a crude stock portfolio database system. There should be at least four data columns: stock ticker symbol, number of shares, purchase price, and current priceyou can add more if you wish, such as percentage gain(loss), 52-week high/low, beta, etc.

Have the user input values for each column to create a single row. Each row should be created as list. Another all-encompassing list will hold all these rows. Once the data is entered, prompt the user for one column to use as the sort metric. Extract the data values of that column into a dictionary as keys, with their corresponding values being the row that contains that key. Be mindful that the sort metric must have non-coincidental keys or else you will lose a row because dictionaries are not allowed to have more than one value with the same key. You may also choose to have additional calculated output, such as percentage gain/loss, current portfolio values, etc.

7-7. *Inverting Dictionaries*. Take a dictionary as input and return one as output, but the values are now the keys and vice versa.

7-8. *Human Resources*. Create a simple name and employee number dictionary application. Have the user enter a list of names and employee numbers. Your interface should allow a sorted output (sorted by name) that displays employee names followed by their employee numbers. Extra credit: Come up with an additional feature that allows for output to be sorted by employee numbers.

7-9. *Translations.*

a.

Create a character translator (that works similar to the Unix `tr` command). This function, which we will call `TR()`, takes three strings as arguments: source, destination, and base strings, and has the following declaration:

```
def tr(srcstr, dststr, string)
```

`srcstr` contains the set of characters you want "translated," `dststr` contains the set of characters to translate to, and `string` is the string to perform the translation on. For example, if `srcstr == 'abc'`, `dststr == 'mno'`, and `string == 'abcdef'`, then `tr()` would output`'mnodef'`. Note that `len(srcstr) == len(dststr)`. For this exercise, you can use the `chr()` and `ord()` BIFs, but they are not necessary to arrive at a solution.

b.

Add a new flag argument to this function to perform case-insensitive translations.

c.

Update your solution so that it can process character deletions. Any extra

characters in `srcstr` that are beyond those that could be mapped to characters in `dststr` should be filtered. In other words, these characters are mapped to no characters in `dststr`, and are thus filtered from the modified string that is returned. For example, if `srcstr` ==`'abcdef'`, `dststr` == `'mno'`, and `string` == `'abcdefghi'`, then `tr()` would output `'mnoghi'`. Note now that `len(srcstr) >= len (dststr)`.

7-10. *Encryption*. Using your solution to the previous problem, and create a "rot13" translator. "rot13" is an old and fairly simplistic encryption routine whereby each letter of the alphabet is rotated 13 characters. Letters in the first half of the alphabet will be rotated to the equivalent letter in the second half and vice versa, retaining case. For example, `a` goes to `n` and `X` goes to `K`. Obviously, numbers and symbols are immune from translation.

(b) Add an application on top of your solution to prompt the user for strings to encrypt (and decrypt on reapplication of the algorithm), as in the following examples:

```
% rot13.py
Enter string to rot13: This is a short sentence.
Your string to en/decrypt was: [This is a short
sentence.].
The rot13 string is: [Guvf vf n fubeg fragrapr.].
%
% rot13.py
Enter string to rot13: Guvf vf n fubeg fragrapr.
Your string to en/decrypt was: [Guvf vf n fubeg
fragrapr.].
The rot13 string is: [This is a short sentence.].
```

7-11. *Definitions*. What constitutes valid dictionary keys? Give examples of valid and invalid dictionary keys.

7-12. *Definitions*. (a) What is a set in the mathematical sense? (b) What is a set type as it relates to Python?

7-13. *Random Numbers*. The next problems use a customization of Exercise 5-17: use `randint()` or `randrange()` in the `random` module to generate a set of numbers: generate between 1 to 10 random numbers numbered randomly between 0 and 9 (inclusive). These values constitute a set A (A can be mutable or otherwise). Create another random set B in a similar manner. Display A | B and A & B each time sets A and B are generated.

7-14. *User Validation*. Alter the previous problem where instead of displaying A | B and A & B, ask the user to input solutions to A | B and A & B, and let the user know if his or her solution was right or wrong. If it is not correct, give the user the ability to correct and revalidate his or her answers. Display the correct results if three incorrect answers are submitted. Extra credit: Use your knowledge of sets to generate potential subsets and ask the user whether they are indeed subsets (or not), and provide corrections and answers as necessary as in the main part of this problem.

7-15. *Set Calculator*. This exercise is inspired by Exercise 12.2 in the free online Java textbook located at http://math.hws.edu/javanotes. Create an application that allows users to input a pair of sets, A and B, and allow users to give an operation symbol, i. e., `in`, `not in`, `&`, `|`, `^`, `<`, `<=`, `>`, `>=`, `==`, `!=`, etc. (For sets, you define the input syntaxthey do not have to be enclosed in brackets as the Java example.) Parse the entire input string and execute the operation on the input sets as requested by the user. Your solution should require fewer lines of Python than the one in Java.

Chapter 8. Conditionals and Loops

Chapter Topics

The primary focus of this chapter are Python's conditional and looping statements, and all their related components. We will take a close look at `if`, `while`, `for`, and their friends `else`, `elif`, `break`, `continue`, and `pass`.

8.1. `if` Statement

The **if** statement for Python will seem amazingly familiar. It is made up of three main components: the keyword itself, an expression that is tested for its truth value, and a code suite to execute if the expression evaluates to non-zero or true. The syntax for an **if** statement is:

```
if expression:
     expr_true_suite
```

The suite of the **if** clause, *expr_true_suite*, will be executed only if the above conditional expression results in a Boolean true value. Otherwise, execution resumes at the next statement following the suite.

8.4.1. Multiple Conditional Expressions

The Boolean operators **and**, **or**, and **not** can be used to provide multiple conditional expressions or perform negation of expressions in the same **if** statement.

```
if not warn and (system_load >= 10):
    print "WARNING: losing resources"
    warn += 1
```

8.1.2. Single Statement Suites

If the suite of a compound statement, i.e., **if** clause, **while** or **for** loop, consists only of a single line, it may go on the same line as the header statement:

```
if make_hard_copy: send_data_to_printer()
```

Single line statements such as the above are valid syntax-wise; however, although it may be convenient, it may make your code more difficult to read, so we recommend you indent the suite on the next line. Another good reason is that if you must add another line to the suite, you have to move that line down anyway.

8.2. `else` Statement

Like other languages, Python features an **`else`** statement that can be paired with an **`if`** statement. The **`else`** statement identifies a block of code to be executed if the conditional expression of the **`if`** statement resolves to a false Boolean value. The syntax is what you expect:

```
if expression:
    expr_true_suite
else:
    expr_false_suite
```

Now we have the obligatory usage example:

```
if passwd == user.passwd:
    ret_str = "password accepted"
    id = user.id
    valid = True
else:
    ret_str = "invalid password entered... try again!"
    valid = False
```

8.2.1. "Dangling `else`" Avoidance

Python's design of using indentation rather than braces for code block delimitation not only helps to enforce code correctness, but it even aids implicitly in avoiding potential problems in code that *is* syntactically correct. One of those such problems is the (in)famous "dangling else" problem, a semantic optical illusion.

We present some C code here to illustrate our example (which is also illuminated by K&R and other programming texts):

```
/* dangling-else in C */
if (balance > 0.00)
    if (((balance - amt) > min_bal) && (atm_cashout() == 1))
      printf("Here's your cash; please take all bills.\n");
else
    printf("Your balance is zero or negative.\n");
```

The question is, which **`if`** does the **`else`** belong to? In the C language, the rule is that the **`else`** stays with the closest **`if`**. In our example above, although indented for the outer **`if`** statement, the **`else`** statement really belongs to the inner **`if`** statement because the C compiler ignores superfluous white space. As a result, if you have a positive balance but it is below the minimum, you will get the horrid (and erroneous) message that your balance is either zero or negative.

Although solving this problem may be easy due to the simplistic nature of the example, any larger sections of code embedded within this framework may be a hair-pulling experience to root out. Python puts up guardrails not necessarily to prevent you from driving off the cliff, but to steer you away from danger. The same example in Python will result in one of the following choices (one of which is correct):

```
if balance > 0.00:
    if balance - amt > min_bal and atm_cashout():
        print "Here's your cash; please take all bills."
else:
    print 'Your balance is zero or negative.'
```

or

```
if balance > 0.00:
    if balance - amt > min_bal and atm_cashout():
        print "Here's your cash; please take all bills."
    else:
        print 'Your balance is zero or negative.'
```

Python's use of indentation forces the proper alignment of code, giving the programmer the ability to make a conscious decision as to which **if** an **else** statement belongs to. By limiting your choices and thus reducing ambiguities, Python encourages you to develop correct code the first time. It is impossible to create a dangling else problem in Python. Also, since parentheses are not required, Python code is easier to read.

PREV NEXT

8.3. `elif` (aka `else-if`) Statement

`elif` is the Python `else-if` statement. It allows one to check multiple expressions for truth value and execute a block of code as soon as one of the conditions evaluates to true. Like the **`else`**, the **`elif`** statement is optional. However, unlike **`else`**, for which there can be at most one statement, there can be an arbitrary number of **`elif`** statements following an **`if`**.

```
if expression1:
   expr1_true_suite
elif expression2:
   expr2_true_suite
               :
elif expressionN:
   exprN_true_suite
else:
   none_of_the_above_suite
```

Proxy for `switch/case` Statement?

At some time in the future, Python may support the `switch` or `case` statement, but you can simulate it with various Python constructs. But even a good number of **`if-elif`** statements are not that difficult to read in Python:

```
if user.cmd == 'create':
    action = "create item"

elif user.cmd == 'delete':
    action = 'delete item'

elif user.cmd == 'update':
    action = 'update item'

else:
    action = 'invalid choice... try again!'
```

Although the above statements do work, you can simplify them with a sequence and the membership operator:

```
if user.cmd in ('create', 'delete', 'update'):
    action = '%s item' % user.cmd
else:
    action = 'invalid choice... try again!'
```

We can create an even more elegant solution using Python dictionaries, which we learned about in Chapter 7, "Mapping and Set Types."

```
msgs = {'create': 'create item',
    'delete': 'delete item',
    'update': 'update item'}
default = 'invalid choice... try again!'
action = msgs.get(user.cmd, default)
```

One well-known benefit of using mapping types such as dictionaries is that the searching is very fast compared to a sequential lookup as in the above **`if-elif-else`** statements or using a **`for`** loop, both of which have to scan the elements one at a time.

PREV NEXT

8.4. Conditional Expressions (aka "the Ternary Operator")

If you are coming from the C/C++ or Java world, it is difficult to ignore or get over the fact that Python has not had a conditional or ternary operator `(C ? X : Y)` for the longest time. (`C` is the conditional expression; `X` represents the resultant expression if `C` is `true` and `Y` if `C` is `False`.) van Rossum Guido has resisted adding such a feature to Python because of his belief in keeping code simple and not giving programmers easy ways to obfuscate their code.

However, after more than a decade, he has given in, mostly because of the error-prone ways in which people have tried to simulate it using **`and`** and **`or`**, many times incorrectly. According to the FAQ, the one way of getting it right is `(C` and `[X]` or `[Y])[0]`. The only problem was that the community could not agree on the syntax. (You really have to take a look at PEP 308 to see all the different proposals.) This is one of the areas of Python in which people have expressed strong feelings.

The final decision came down to van Rossum Guido choosing the most favored (and his most favorite) of all the choices, then applying it to various modules in the standard library. According to the PEP, "this review approximates a sampling of real-world use cases, across a variety of applications, written by a number of programmers with diverse backgrounds." And this is the syntax that was finally chosen for integration into Python 2.5: `X` **`if`** `C` **`else`** `Y`.

The main motivation for even having a ternary operator is to allow the setting of a value based on a conditional all on a single line, as opposed to the standard way of using an **`if-else`** statement, as in this `min()` example using numbers x and y:

```
>>> x, y = 4, 3
>>> if x < y:
...     smaller = x
... else:
...     smaller = y
...
>>> smaller
3
```

In versions prior to 2.5, Python programmers at best could do this:

```
>>> smaller = (x < y and [x] or [y])[0]
>>> smaller
3
```

In versions 2.5 and newer, this can be further simplified to:

```
>>> smaller = x if x < y else y
>>> smaller
3
```

8.5. `while` Statement

Python's **`while`** is the first looping statement we will look at in this chapter. In fact, it is a conditional looping statement. In comparison with an **`if`** statement where a true expression will result in a single execution of the **`if`** clause suite, the suite in a **`while`** clause will be executed continuously in a loop until that condition is no longer satisfied.

8.5.1. General Syntax

Here is the syntax for a **`while`** loop:

```
while expression:
    suite_to_repeat
```

The `suite_to_repeat` clause of the **`while`** loop will be executed continuously in a loop until *`expression`* evaluates to Boolean `False`. This type of looping mechanism is often used in a counting situation, such as the example in the next subsection.

8.5.2. Counting Loops

```
count = 0
while (count < 9):
    print 'the index is:', count
    count += 1
```

The suite here, consisting of the **`print`** and increment statements, is executed repeatedly until `count` is no longer less than 9. With each iteration, the current value of the index `count` is displayed and then bumped up by 1. If we take this snippet of code to the Python interpreter, entering the source and seeing the resulting execution would look something like:

```
>>> count = 0
>>> while (count < 9):
...     print 'the index is:', count
...     count += 1
...
the index is: 0
the index is: 1
the index is: 2
the index is: 3
the index is: 4
the index is: 5
the index is: 6
the index is: 7
the index is: 8
```

8.5.3. Infinite Loops

One must use caution when using **while** loops because of the possibility that the condition never resolves to a false value. In such cases, we would have a loop that never ends on our hands. These "infinite" loops are not necessarily bad thingsmany communications "servers" that are part of client/server systems work exactly in that fashion. It all depends on whether or not the loop was meant to run forever, and if not, whether the loop has the possibility of terminating; in other words, will the expression ever be able to evaluate to false?

```
while True:
    handle, indata = wait_for_client_connect()
    outdata = process_request(indata)
    ack_result_to_client(handle, outdata)
```

For example, the piece of code above was set deliberately to never end because `TRue` is not going to somehow change to `False`. The main point of this server code is to sit and wait for clients to connect, presumably over a network link. These clients send requests which the server understands and processes.

After the request has been serviced, a return value or data is returned to the client who may either drop the connection altogether or send another request. As far as the server is concerned, it has performed its duty to this one client and returns to the top of the loop to wait for the next client to come along. You will find out more about client/server computing in Chapter 16, "Network Programming" and Chapter 17, "Internet Client Programming."

PREV NEXT

8.6. for Statement

The other looping mechanism in Python comes to us in the form of the **for** statement. It represents the single most powerful looping construct in Python. It can loop over sequence members, it is used in list comprehensions and generator expressions, and it knows how to call an iterator's `next()` method and gracefully ends by catching `StopIteration` exceptions (all under the covers). If you are new to Python, we will tell you now that you will be using **for** statements a lot.

Unlike the traditional conditional looping **for** statement found in mainstream languages like C/C++, Fortran, or Java, Python's **for** is more akin to a shell or scripting language's iterative **foreach** loop.

8.6.1. General Syntax

The **for** loop traverses through individual elements of an iterable (like a sequence or iterator) and terminates when all the items are exhausted. Here is its syntax:

```
for iter_var in iterable:
    suite_to_repeat
```

With each loop, the `iter_var` iteration variable is set to the current element of the iterable (sequence, iterator, or object that supports iteration), presumably for use in `suite_to_repeat`.

8.6.2. Used with Sequence Types

In this section, we will see how the **for** loop works with the different sequence types. The examples will include string, list, and tuple types.

```
>>> for each Letter in 'Names':
...     print 'current letter:', each Letter
...
current letter: N
current letter: a
current letter: m
current letter: e
current letter: s
```

When iterating over a string, the iteration variable will always consist of only single characters (strings of length 1). Such constructs may not necessarily be useful. When seeking characters in a string, more often than not, the programmer will either use **in** to test for membership, or one of the string module functions or string methods to check for sub strings.

One place where seeing individual characters does come in handy is during the debugging of sequences in a **for** loop in an application where you are expecting strings or entire objects to show up in your **print** statements. If you see individual characters, this is usually a sign that you received a single string rather than a sequence of objects.

There are three basic ways of iterating over a sequence:

Iterating by Sequence Item

```
>>> nameList = ['Walter', "Nicole", 'Steven', 'Henry']
>>> for eachName in nameList:
...     print eachName, "Lim"
...
Walter Lim
Nicole Lim
Steven Lim
Henry Lim
```

In the above example, a list is iterated over, and for each iteration, the `eachName` variable contains the list element that we are on for that particular iteration of the loop.

Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself:

```
>>> nameList = ['Cathy', "Terry", 'Joe', 'Heather', 'Lucy']
>>> for nameIndex in range(len(nameList)):
...     print "Liu,", nameList[nameIndex]
...
Liu, Cathy
Liu, Terry
Liu, Joe
Liu, Heather
Liu, Lucy
```

Rather than iterating through the elements themselves, we are iterating through the indices of the list.

We employ the assistance of the `len()` built-in function, which provides the total number of elements in the tuple as well as the `range()` built-in function (which we will discuss in more detail below) to give us the actual sequence to iterate over.

```
>>> len(nameList)
5
>>> range(len(nameList))
[0, 1, 2, 3, 4]
```

Using `range()`, we obtain a list of the indexes that `nameIndex` iterates over; and using the slice/subscript operator (`[ ]`), we can obtain the corresponding sequence element.

Those of you who are performance pundits will no doubt recognize that iteration by sequence item wins over iterating via index. If not, this is something to think about. (See Exercise 8-13.)

Iterating with Item and Index

2.3

The best of both worlds comes from using the `enumerate()` built-in function, which was added to Python in version 2.3. Enough said ... here is some code:

```
>>> nameList = ['Donn', 'Shirley', 'Ben', 'Janice',
...     'David', 'Yen', 'Wendy']
>>> for i, eachLee in enumerate(nameList):
...     print "%d %s Lee" % (i+1, eachLee)
...
1 Donn Lee
2 Shirley Lee

3 Ben Lee
4 Janice Lee
5 David Lee
6 Yen Lee
7 Wendy Lee
```

8.6.3. Used with Iterator Types

Using **`for`** loops with iterators is identical to using them with sequences. The only difference is that the **`for`** statement must do a little bit of extra work on your behalf. An iterator does not represent a set of items to loop over.

Instead, iterator objects have a `next()` method, which is called to get subsequent items. When the set of items has been exhausted, the iterator raises the `StopIteration` exception to signal that it has finished. Calling `next()` and catching `StopIteration` is built-in to the **`for`** statement.

When you are using a **`for`** loop with an iterator, the code is nearly identical to that of looping over sequence items. In fact, for most cases, you cannot tell that you are iterating over a sequence or an iterator, hence the reason why you will see us refer to iterating over an iterable, which could mean a sequence, an iterator, or any object that supports iteration, e.g., has a `next()` method.

8.6.4. **`range()`** Built-in Function

We mentioned above during our introduction to Python's **`for`** loop that it is an iterative looping mechanism. Python also provides a tool that will let us use the **`for`** statement in a traditional pseudo-conditional setting, i.e., when counting from one number to another and quitting once the final number has been reached or some condition is no longer satisfied.

The built-in function `range()` can turn your `foreach`-like **`for`** loop back into one that you are more familiar with, i.e., counting from 0 to 10, or counting from 10 to 100 in increments of 5.

`range()` Full Syntax

Python presents two different ways to use `range()`. The full syntax requires that two or all three integer arguments are present:

```
range(start, end, step=1)
```

`range()` will then return a list where for any *k*, *start* <= *k* < *end* and *k* iterates from *start* to *end* in increments of *step*. *step* cannot be 0, or an error condition will occur.

```
>>> range(2, 19, 3)
[2, 5, 8, 11, 14, 17]
```

If *step* is omitted and only two arguments given, *step* takes a default value of 1.

```
>>> range(3, 7)
[3, 4, 5, 6]
```

Let's take a look at an example used in the interpreter environment:

```
>>> for eachVal in range(2, 19, 3):
...       print "value is:", eachVal
...
value is: 2
value is: 5
value is: 8
value is: 11
value is: 14
value is: 17
```

Our **for** loop now "counts" from 2 to 19, incrementing by steps of 3. If you are familiar with C, then you will notice the direct correlation between the arguments of `range()` and those of the variables in the C **for** loop:

```
/* equivalent loop in C */
for (eachVal = 2; eachVal < 19; eachVal += 3) {
   printf("value is: %d\n", eachVal);
}
```

Although it seems like a conditional loop now (checking if `eachVal < 19`), reality tells us that `range()` takes our conditions and generates a list that meets our criteria, which in turn is used by the same Python **for** statement.

range() Abbreviated Syntax

`range()` also has two abbreviated syntax formats:

```
range(end)

range(start, end)
```

We saw the shortest syntax earlier in Chapter 2. Given only a single value, *start* defaults to 0, *step* defaults to 1, and `range()` returns a list of numbers from zero up to the argument `end`:

```
>>> range(5)
[0, 1, 2, 3, 4]
```

Given two values, this midsized version of `range()` is exactly the same as the long version of `range()` taking two parameters with `step` defaulting to 1. We will now take this to the Python interpreter and plug in **`for`** and **`print`** statements to arrive at:

```
>>> for count in range(2, 5):
...     print count
...
2
3
4
```

Core Note: Why not just one syntax for `range()`?

Now that you know both syntaxes for `range()`, one nagging question you may have is, why not just combine the two into a single one that looks like this?

```
range(start=0, end, step=1) # invalid
```

This syntax will work for a single argument or all three, but not two. It is illegal because the presence of `step` requires `start` to be given. In other words, you cannot provide `end` and `step` in a two-argument version because they will be (mis)interpreted as `start` and `end`.

8.6.5. `xrange()` Built-in Function

`xrange()` is similar to `range()` except that if you have a really large range list, `xrange()` may come in handier because it does not have to make a complete copy of the list in memory. This built-in was made for exclusive use in **`for`** loops. It does not make sense outside a **`for`** loop. Also, as you can imagine, the performance will not be as good because the entire list is *not* in memory. In future versions of Python, `range()` will eventually become like `xrange()`, returing an iterable object (not a list nor an iterator though)it will be similar to views as discussed in the previous chapter.

8.6.6. Sequence-Related Built-in Functions

sorted(), reversed(), enumerate(), zip()

Below are some examples of using these loop-oriented sequence-related functions. The reason why they

are "sequence-related" is that half of them (`sorted()` and `zip()`) return a real sequence (list), while the other two (`reversed()` and `enumerate()`) return iterators (sequence-like).

```
>>> albums = ('Poe', 'Gaudi', 'Freud', 'Poe2')
>>> years = (1976, 1987, 1990, 2003)
>>> for album in sorted(albums):
...     print album,
...
Freud Gaudi Poe Poe2
>>>
>>> for album in reversed(albums):
...     print album,
...
Poe2 Freud Gaudi Poe
>>>
>>> for i, album in enumerate(albums):
...     print i, album
...
0 Poe
1 Gaudi
2 Freud
3 Poe2
>>>
>>> for album, yr in zip(albums, years):
...     print yr, album
...
1976 Poe
1987 Gaudi
1990 Freud
2003 Poe2
```

Now that we have covered all the loops Python has to offer, let us take a look at the peripheral commands that typically go together with loops. These include statements to abandon the loop (**`break`**) and to immediately begin the next iteration (**`continue`**).

PREV NEXT

8.7. `break` Statement

The **`break`** statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional **`break`** found in C. The most common use for **`break`** is when some external condition is triggered (usually by testing with an **`if`** statement), requiring a hasty exit from a loop. The **`break`** statement can be used in both **`while`** and **`for`** loops.

```
count = num / 2
while count > 0:
    if num % count == 0:
        print count, 'is the largest factor of', num
        break
    count -= 1
```

The task of this piece of code is to find the largest divisor of a given number `num`. We iterate through all possible numbers that could possibly be factors of `num`, using the `count` variable and decrementing for every value that does *not* divide `num`. The first number that evenly divides `num` is the largest factor, and once that number is found, we no longer need to continue and use **`break`** to terminate the loop.

```
phone2remove = '555-1212'
for eachPhone in phoneList:
    if eachPhone == phone2remove:
        print "found", phone2remove, '... deleting'
        deleteFromPhoneDB(phone2remove)
        break
```

The **`break`** statement here is used to interrupt the iteration of the list. The goal is to find a target element in the list, and, if found, to remove it from the database and break out of the loop.

8.8. `continue` Statement

Core Note: `continue` statements

Whether in Python, C, Java, or any other structured language that features the **`continue`** *statement, there is a misconception among some beginning programmers that the traditional* **`continue`** *statement "immediately starts the next iteration of a loop." While this may seem to be the apparent action, we would like to clarify this somewhat invalid supposition. Rather than beginning the next iteration of the loop when a* **`continue`** *statement is encountered, a* **`continue`** *statement terminates or discards the remaining statements in the current loop iteration and goes back to the top. If we are in a conditional loop, the conditional expression is checked for validity before beginning the next iteration of the loop. Once confirmed, then the next iteration begins. Likewise, if the loop were iterative, a determination must be made as to whether there are any more arguments to iterate over. Only when that validation has completed successfully can we begin the next iteration.*

The **`continue`** statement in Python is not unlike the traditional **`continue`** found in other high-level languages. The **`continue`** statement can be used in both **`while`** and **`for`** loops. The **`while`** loop is conditional, and the **`for`** loop is iterative, so using **`continue`** is subject to the same requirements (as highlighted in the Core Note above) before the next iteration of the loop can begin. Otherwise, the loop will terminate normally.

```
valid = False
count = 3
while count > 0:
    input = raw_input("enter password")
    # check for valid passwd
    for eachPasswd in passwdList:
        if input == eachPasswd:
            valid = True
            break
    if not valid:    # (or valid == 0)
        print "invalid input"
        count -= 1
        continue
    else:
        break
```

In this combined example using **`while`**, **`for`**, **`if`**, **`break`**, and **`continue`**, we are looking at validating user input. The user is given three opportunities to enter the correct password; otherwise, the `valid` variable remains a false value of 0, which presumably will result in appropriate action being taken soon after.

8.9. pass Statement

One Python statement not found in C is the **pass** statement. Because Python does not use curly braces to delimit blocks of code, there are places where code is syntactically required. We do not have the equivalent empty braces or single semicolon the way C does to indicate "do nothing." If you use a Python statement that expects a sub-block of code or suite, and one is not present, you will get a syntax error condition. For this reason, we have **pass**, a statement that does absolutely nothingit is a true NOP, to steal the "No OPeration" assembly code jargon. Style- and development-wise, **pass** is also useful in places where your code will eventually go, but has not been written yet (in stubs, for example):

```
def foo_func():
   pass
```

or

```
if user_choice == 'do_calc':
   pass
else:
   pass
```

This code structure is helpful during the development or debugging stages because you want the structure to be there while the code is being created, but you do not want it to interfere with the other parts of the code that have been completed already. In places where you want nothing to execute, **pass** is a good tool to have in the box.

Another popular place is with exception handling, which we will take a look at in Chapter 10; this is where you can track an error if it occurs, but take no action if it is not fatal (you just want to keep a record of the event or perform an operation under the covers if an error occurs).

8.10. `else` Statement ... Take Two

In C (as well as in most other languages), you will *not* find an **`else`** statement outside the realm of conditional statements, yet Python bucks the trend again by offering these in **`while`** and **`for`** loops. How do they work? When used with loops, an **`else`** clause will be executed only if a loop finishes to completion, meaning they were not abandoned by **`break`**.

One popular example of **`else`** usage in a **`while`** statement is in finding the largest factor of a number. We have implemented a function that performs this task, using the **`else`** statement with our **`while`** loop. The `showMaxFactor()` function in Example 8.1 (`maxFact.py`) utilizes the **`else`** statement as part of a **`while`** loop.

Example 8.1. `while-else` Loop Example (`maxFact.py`)

This program displays the largest factors for numbers between 10 and 20. If the number is prime, the script will indicate that as well.

```
1   #!/usr/bin/env python
2
3   def showMaxFactor(num):
4       count = num / 2
5       while count > 1:
6           if num % count == 0:
7               print 'largest factor of %d is %d' % \
8                   (num, count)
9               break
10          count -= 1
11      else:
12          print num, "is prime"
13
14   for eachNum in range(10, 21):
15       showMaxFactor(eachNum)
```

The loop beginning on line 3 in `showMaxFactor()` counts down from half the amount (starts checking if two divides the number, which would give the largest factor). The loop decrements each time (line 10) through until a divisor is found (lines 6-9). If a divisor has not been found by the time the loop decrements to 1, then the original number must be prime. The **`else`** clause on lines 11-12 takes care of this case. The main part of the program on lines 14-15 fires off the requests to `showMaxFactor()` with the numeric argument.

Running our program results in the following output:

```
largest factor of 10 is 5
11 is prime
largest factor of 12 is 6
13 is prime
```

```
largest factor of 14 is 7
largest factor of 15 is 5
largest factor of 16 is 8
17 is prime
largest factor of 18 is 9
19 is prime
largest factor of 20 is 10
```

Likewise, a **for** loop can have a post-processing **else**. It operates exactly the same way as for a **while** loop. As long as the **for** loop exits normally (not via **break**), the **else** clause will be executed. We saw such an example in Section 8.5.3.

Table 8.1 summarizes with which conditional or looping statements auxiliary statements can be used.

Table 8.1. Auxiliary Statements to Loops and Conditionals

	Loops and Conditionals		
Auxiliary Statements	**if**	**while**	**for**
elif	•		
else	•	•	•
break		•	•
continue		•	•
pass[a]	•	•	•

[a] *pass* is valid anywhere a suite (single or multiple statements) is required (also includes **elif**, **else**, **class**, **def**, **TRy**, **except**, **finally**).

8.11. Iterators and the `iter()` Function

8.11.1. What Are Iterators?

Iterators were added to Python in version 2.2 to give sequence-like objects a sequence-like interface. We formally introduced sequences back in Chapter 6. They are just data structures that you can "iterate" over by using their index starting at 0 and continuing till the final item of the sequence. Because you can do this "counting," iterating over sequences is trivial. Iteration support in Python works seamlessly with sequences but now also allows programmers to iterate through non-sequence types, including user-defined objects.

Iterators come in handy when you are iterating over something that is not a sequence but exhibits behavior that makes it *seem* like a sequence, for example, keys of a dictionary, lines of a file, etc. When you use loops to iterate over an object item, you will not be able to easily tell whether it is an iterator or a sequence. The best part is that you do not have to care because Python makes it seem like a sequence.

8.11.2. Why Iterators?

The defining PEP (234) cites that iterators:

- Provide an extensible iterator interface.
- Bring performance enhancements to list iteration.
- Allow for big performance improvements in dictionary iteration.
- Allow for the creation of a true iteration interface as opposed to overriding methods originally meant for random element access.
- Be backward-compatible with all existing user-defined classes and extension objects that emulate sequences and mappings.
- Result in more concise and readable code that iterates over non-sequence collections (mappings and files, for instance).

8.11.3. How Do You Iterate?

Basically, instead of an index to count sequentially, an iterator is any item that has a `next()` method. When the next item is desired, either you or a looping mechanism like **`for`** will call the iterators `next()` method to get the next value. Once the items have been exhausted, a `StopIteration` exception is raised, not to indicate an error, but to let folks know that we are done.

Iterators do have some restrictions, however. For example, you cannot move backward, go back to the beginning, or copy an iterator. If you want to iterate over the same objects again (or simultaneously), you have to create another iterator object. It isn't all that bad, however, as there are various tools to help you with using iterators.

2.5

There is a `reversed()` built-in function that returns an iterator that traverses an iterable in reverse order. The `enumerate()` BIF also returns an iterator. Two new BIFs, `any()` and `all()`, made their debut in Python 2.5they will return `true` if any or all items traversed across an iterator have a Boolean `true` value, respectively. We saw earlier in the chapter how you can use it in a **`for`** loop to iterate over both the index and the item of an iterable. There is also an entire module called `itertools` that contains various iterators you may find useful.

8.11.4. Using Iterators with ...

Sequences

As mentioned before, iterating through Python sequence types is as expected:

```
>>> myTuple = (123, 'xyz', 45.67)
>>> i = iter(myTuple)
>>> i.next()
123
>>> i.next()
'xyz'
>>> i.next()
45.67
>>> i.next()
Traceback (most recent call last):
  File "", line 1, in ?
StopIteration
```

If this had been an actual program, we would have enclosed the code inside a **`try-except`** block. Sequences now automatically produce their own iterators, so a **`for`** loop:

```
for i in seq:
    do_something_to(i)
```

under the covers now really behaves like this:

```
fetch = iter(seq)
while True:
    try:
        i = fetch.next()
    except StopIteration:
        break
    do_something_to(i)
```

However, your code does not need to change because the **`for`** loop itself calls the iterator's `next()` method (as well as monitors for `StopIteration`).

Dictionaries

Dictionaries and files are two other Python data types that received the iteration makeover. A dictionary's iterator traverses its keys. The idiom `for eachKey in myDict.keys()` can be shortened to `for eachKey in myDict` as shown here:

```
>>> legends = { ('Poe', 'author'): (1809, 1849, 1976),
...  ('Gaudi', 'architect'): (1852, 1906, 1987),
...  ('Freud', 'psychoanalyst'): (1856, 1939, 1990)
... }
...
>>> for eachLegend in legends:
...    print 'Name: %s\tOccupation: %s' % eachLegend
...    print '  Birth: %s\tDeath: %s\tAlbum: %s\n' \
...    % legends[eachLegend]
...
Name: Freud     Occupation: psychoanalyst
  Birth: 1856   Death: 1939     Album: 1990

Name: Poe       Occupation: author
  Birth: 1809   Death: 1849     Album: 1976

Name: Gaudi     Occupation: architect
  Birth: 1852   Death: 1906     Album: 1987
```

In addition, three new built-in dictionary methods have been introduced to define the iteration: `myDict.iterkeys()` (iterate through the keys), `myDict.itervalues()` (iterate through the values), and `myDict.iteritems()` (iterate through key/value pairs). Note that the `in` operator has been modified to check a dictionary's keys. This means the Boolean expression `myDict.has_key(anyKey)` can be simplified as `anyKey in myDict`.

Files

File objects produce an iterator that calls the `readline()` method. Thus, they loop through all lines of a text file, allowing the programmer to replace essentially `for eachLine in myFile.readlines()` with the more simplistic `for eachLine in myFile`:

```
>>> myFile = open('config-win.txt')
>>> for eachLine in myFile:
...     print eachLine,   # comma suppresses extra \n
...
[EditorWindow]
font-name: courier new
font-size: 10
>>> myFile.close()
```

8.11.5. Mutable Objects and Iterators

Remember that interfering with mutable objects while you are iterating them is not a good idea. This was a problem before iterators appeared. One popular example of this is to loop through a list and remove items from it if certain criteria are met (or not):

```
for eachURL in allURLs:
    if not eachURL.startswith('http://'):
        allURLs.remove(eachURL)       # YIKES!!
```

All sequences are immutable except lists, so the danger occurs only there. A sequence's iterator only keeps track of the Nth element you are on, so if you change elements around during iteration, those updates will be reflected as you traverse through the items. If you run out, then `StopIteration` will be raised.

When iterating through keys of a dictionary, you must not modify the dictionary. Using a dictionary's `keys()` method is okay because `keys()` returns a list that is independent of the dictionary. But iterators are tied much more intimately with the actual object and will not let us play that game anymore:

```
>>> myDict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> for eachKey in myDict:
...    print eachKey, myDict[eachKey]
...    del myDict[eachKey]
...
a 1
Traceback (most recent call last):
  File "", line 1, in ?
RuntimeError: dictionary changed size during iteration
```

This will help prevent buggy code. For full details on iterators, see PEP 234.

8.11.6. How to Create an Iterator

You can take an item and call `iter()` on it to turn it into an iterator. Its syntax is one of the following:

```
iter(obj)
iter(func, sentinel)
```

If you call `iter()` with one object, it will check if it is just a sequence, for which the solution is simple: It will just iterate through it by (integer) index from 0 to the end. Another way to create an iterator is with a class. As we will see in Chapter 13, a class that implements the `__iter__()` and `next()` methods can be used as an iterator.

If you call `iter()` with two arguments, it will repeatedly call *func* to obtain the next value of iteration until that value is equal to *sentinel*.

8.12. List Comprehensions

List comprehensions (or "list comps" for short) come to us from the functional programming language Haskell. They are an extremely valuable, simple, and flexible utility tool that helps us create lists on the fly. They were added to Python in version 2.0.

Up ahead in Functions (Chapter 11), we will be discussing long-time Python functional programming features like `lambda`, `map()`, and `filter()`. They have been around in Python for quite a while, but with list comprehensions, they have simplified their use to only requiring a list comp instead. `map()` is a function that applies an operation to list members, and `filter()` filters out list members based on a conditional expression. Finally, `lambda` allows you to create one-line function objects on the fly. It is not important that you learn them now, but you will see examples of them in this section because we are discussing the merits of list comps. Let us take a look at the simpler list comprehension syntax first:

```
[expr for iter_var in iterable]
```

The core of this statement is the `for` loop, which iterates over each item of *iterable*. The prefixed *expr* is applied for each member of the sequence, and the resulting values comprise the list that the expression yields. The iteration variable need not be part of the expression.

Here is a sneak preview of some code from Chapter 11. It has a `lambda` function that squares the members of a sequence:

```
>>> map(lambda x: x ** 2, range(6))
[0, 1, 4, 9, 16, 25]
```

We can replace this code with the following list comprehension statement:

```
>>> [x ** 2 for x in range(6)]
[0, 1, 4, 9, 16, 25]
```

In the new statement, only one function call (`range()`) is made (as opposed to three`range()`, `map()`, and the `lambda` function). You may also use parentheses around the expression if `[(x ** 2) for x in range (6)]` is easier for you to read. This syntax for list comprehensions can be a substitute for and is more efficient than using the `map()` built-in function along with `lambda`.

List comprehensions also support an extended syntax with the `if` statement:

```
[expr for iter_var in iterable if cond_expr]
```

This syntax will filter or "capture" sequence members only if they meet the condition provided for in the *cond_expr* conditional expression during iteration.

Recall the following `odd()` function below, which determines whether a numeric argument is odd or even (returning 1 for odd numbers and 0 for even numbers):

```
def odd(n):
    return n % 2
```

We were able to take the core operation from this function, and use it with `filter()` and **`lambda`** to obtain the set of odd numbers from a sequence:

```
>>> seq = [11, 10, 9, 9, 10, 10, 9, 8, 23, 9, 7, 18, 12, 11, 12]
>>> filter(lambda x: x % 2, seq)
[11, 9, 9, 9, 23, 9, 7, 11]
```

As in the previous example, we can bypass the use of `filter()` and **`lambda`** to obtain the desired set of numbers with list comprehensions:

```
>>> [x for x in seq if x % 2]
[11, 9, 9, 9, 23, 9, 7, 11]
```

Let us end this section with a few more practical examples.

Matrix Example

Do you want to iterate through a matrix of three rows and five columns? It is as easy as:

```
>>> [(x+1,y+1) for x in range(3) for y in range(5)]
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 1), (2, 2), (2,
 3), (2, 4), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5)]
```

Disk File Example

Now let us say we have the following data file and want to count the total number of non-whitespace characters in the file `hhga.txt`:

```
And the Lord spake, saying, "First shalt thou take out the
Holy Pin. Then shalt thou count to three, no more, no less.
Three shall be the number thou shalt count, and the number of
the counting shall be three. Four shalt thou not count,
neither count thou two, excepting that thou then proceed to
three. Five is right out. Once the number three, being the
third number, be reached, then lobbest thou thy Holy Hand
Grenade of Antioch towards thy foe, who, being naughty in My
sight, shall snuff it."
```

We know that we can iterate through each line with for line in data, but more than that, we can also go

and split each line up into words, and we can sum up the number of words to get a total like this:

```
>>> f = open('hhga.txt', 'r')
>>> len([word for line in f for word in line.split()])
91
```

Let us get a quick total file size:

```
import os
>>> os.stat('hhga.txt').st_size
499L
```

Assuming that there is at least one whitespace character in the file, we know that there are fewer than 499 *non*-whitespace characters in the file. We can sum up the length of each word to arrive at our total:

```
>>> f.seek(0)
>>> sum([len(word) for line in f for word in line.split()])
408
```

Note we have to rewind back to the beginning of the file each time through because the iterator exhausts it. But wow, a non-obfuscated one-liner now does something that used to take many lines of code to accomplish!

As you can see, list comps support multiple nested **for** loops and more than one **if** clause. The full syntax can be found in the official documentation. You can also read more about list comprehensions in PEP 202.

PREV NEXT

8.13. Generator Expressions

Generator expressions extend naturally from list comprehensions ("list comps"). When list comps came into being in Python 2.0, they revolutionized the language by giving users an extremely flexible and expressive way to designate the contents of a list on a single line. Ask any long-time Python user what new features have changed the way they program Python, and list comps should be near the top of the list.

Another significant feature that was added to Python in version 2.2 was the generator. A generator is a specialized function that allows you to return a value and "pause" the execution of that code and resume it at a later time. We will discuss generators in Chapter 11.

2.0-2.4

The one weakness of list comps is that all of the data have to be made available in order to create the entire list. This can have negative consequences if an iterator with a large dataset is involved. Generator expressions resolve this issue by combining the syntax and flexibility of list comps with the power of generators.

Introduced in Python 2.4, generator expressions are similar to list comprehensions in that the basic syntax is nearly identical; however, instead of building a list with values, they return a generator that "yields" after processing each item. Because of this, generator expressions are much more memory efficient by performing "lazy evaluation." Take a look at how similar they appear to list comps:

LIST COMPREHENSION:

```
[expr for iter_var in iterable if cond_expr]
```

GENERATOR EXPRESSION:

```
(expr for iter_var in iterable if cond_expr)
```

Generator expressions do not make list comps obsolete. They are just a more memory-friendly construct, and on top of that, are a great use case of generators. We now present a set of generator expression examples, including a long-winded one at the end showing you how Python code has changed over the years.

Disk File Example

In the previous section on list comprehensions, we took a look at finding the total number of non-whitespace characters in a text file. In the final snippet of code, we showed you how to perform that in one line of code using a list comprehension. If that file became unwieldy due to size, it would become fairly unfriendly memory-wise because we would have to put together a very long list of word lengths.

Instead of creating that large list, we can use a generator expression to perform the summing. Instead

of building up this long list, it will calculate individual lengths and feed it to the `sum()` function, which takes not just lists but also iterables like generator expressions. We can then shorten our example above to be even more optimal (code- and execution-wise):

```
>>> sum(len(word) for line in data for word in line.split())
408
```

All we did was remove the enclosing list comprehension square brackets: Two bytes shorter and it saves memory ... very environmentally friendly!

Cross-Product Pairs Example

Generator expressions are like list comprehensions in that they are lazy, which is their main benefit. They are also great ways of dealing with other lists and generators, like `rows` and `cols` here:

```
rows = [1, 2, 3, 17]

def cols():         # example of simple generator
    yield 56
    yield 2
    yield 1
```

We do not need to create a new list. We can piece together things on the fly. Let us create a generator expression for `rows` and `cols`:

```
x_product_pairs = ((i, j) for i in rows for j in cols())
```

Now we can loop through `x_product_pairs`, and it will loop through `rows` and `cols` lazily:

```
>>> for pair in x_product_pairs:
...     print pair
...
(1, 56)
(1, 2)
(1, 1)
(2, 56)
(2, 2)
(2, 1)
(3, 56)
(3, 2)
(3, 1)

(17, 56)
(17, 2)
(17, 1)
```

Refactoring Example

Let us look at some evolutionary code via an example that finds the longest line in a file. In the old

days, the following was acceptable for reading a file:

```
f = open('/etc/motd', 'r')
longest = 0
while True:
    linelen = len(f.readline().strip())
    if not linelen: break
    if linelen > longest:
        longest = linelen
f.close()
return longest
```

Actually, this is not *that* old. If it were really old Python code, the Boolean constant `TRue` would be the integer one, and instead of using the string `strip()` method, you would be using the `string` module:

```
import string
          :
    len(string.strip(f.readline()))
```

Since that time, we realized that we could release the (file) resource sooner if we read all the lines at once. If this was a log file used by many processes, then it behooves us not to hold onto a (write) file handle for an extended period of time. Yes, our example is for read, but you get the idea. So the preferred way of reading in lines from a file changed slightly to reflect this preference:

```
f = open('/etc/motd', 'r')
longest = 0
allLines = f.readlines()
f.close()
for line in allLines:
    linelen = len(line.strip())
    if linelen > longest:
        longest = linelen
return longest
```

List comps allow us to simplify our code a little bit more and give us the ability to do more processing before we get our set of lines. In the next snippet, in addition to reading in the lines from the file, we call the string `strip()` method immediately instead of waiting until later.

```
f = open('/etc/motd', 'r')
longest = 0
allLines = [x.strip() for x in f.readlines()]
f.close()
for line in allLines:
    linelen = len(line)
    if linelen > longest:
        longest = linelen
return longest
```

Still, both examples above have a problem when dealing with a large file as `readlines()` reads in all its lines. When iterators came around, and files became their own iterators, `readlines()` no longer needed to be called. While we are at it, why can't we just make our data set the set of line *lengths* (instead of

lines)? That way, we can use the `max()` built-in function to get the longest string length:

```
f = open('/etc/motd', 'r')
allLineLens = [len(x.strip()) for x in f]
f.close()
return max(allLineLens)
```

The only problem here is that even though you are iterating over `f` line by line, the list comprehension itself needs all lines of the file read into memory in order to generate the list. Let us simplify our code even more: we will replace the list comp with a generator expression and move it inside the call to `max()` so that all of the complexity is on a single line:

```
f = open('/etc/motd', 'r')
longest = max(len(x.strip()) for x in f)
f.close()
return longest
```

One more refactoring, which we are not as much fans of, is dropping the file mode (defaulting to read) and letting Python clean up the open file. It is not as bad as if it were a file open for write, however, but it does work:

```
return max(len(x.strip()) for x in open('/etc/motd'))
```

We have come a long way, baby. Note that even a one-liner is not obfuscated enough in Python to make it difficult to read. Generator expressions were added in Python 2.4, and you can read more about them in PEP 289.

8.14. Related Modules

Iterators were introduced in Python 2.2, and the `itertools` module was added in the next release (2.3) to aid developers who had discovered how useful iterators were but wanted some helper tools to aid in their development. The interesting thing is that if you read the documentation for the various utilities in itertools, you will discover generators. So there is a relationship between iterators and generators. You can read more about this relationship in Chapter 11, "Functions".

8.15. Exercises

8-1. *Conditionals*. Study the following code:

```
# statement A
if x > 0:
    # statement B
    pass

elif x < 0:
    # statement C
    pass

else:
    # statement D
    pass

# statement E
```

a.

Which of the statements above (A, B, C, D, E) will be executed if x < 0?

b.

Which of the statements above will be executed if x = = 0?

c.

Which of the statements above will be executed if x > 0?

8-2. *Loops*. Write a program to have the user input three (3) numbers: (**f**)rom, (**t**)o, and (**i**)ncrement. Count from **f** to **t** in increments of **i**, *inclusive* of **f** and **t**. For example, if the input is **f** == 2, **t** == 26, and **i** == 4, the program would output: 2, 6, 10, 14, 18, 22, 26.

8-3. *range()*. What argument(s) could we give to the `range()` built-in function if we wanted the following lists to be generated?

a.

`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

b.

`[3, 6, 9, 12, 15, 18]`

c.

`[-20, 200, 420, 640, 860]`

8-4. *Prime Numbers*. We presented some code in this chapter to determine a number's largest factor or if it is prime. Turn this code into a Boolean function called `isprime()` such that the input is a single value, and the result returned is `true` if the number is prime and `False` otherwise.

8-5. *Factors*. Write a function called `getfactors()` that takes a single integer as an argument and returns a list of all its factors, including 1 and itself.

8-6. *Prime Factorization*. Take your solutions for `isprime()` and `getfactors()` in the previous problems and create a function that takes an integer as input and returns a list of its prime factors. This process, known as *prime factorization*, should output a list of factors such that if multiplied together, they will result in the original number. Note that there could be repeats in the list. So if you gave an input of 20, the output would be [2, 2, 5].

8-7. *Perfect Numbers*. A perfect number is one whose factors (except itself) sum to itself. For example, the factors of 6 are 1, 2, 3, and 6. Since 1 + 2 + 3 is 6, it (6) is considered a perfect number. Write a function called `isperfect()` which takes a single integer input and outputs 1 if the number is perfect and 0 otherwise.

8-8. *Factorial*. The factorial of a number is defined as the product of all values from one to that number. A shorthand for N factorial is N! where N! == factorial(N) == 1 * 2 * 3 * ... * (N-2) * (N-1) * N. So 4! == 1 * 2 * 3 * 4. Write a routine such that given N, the value N! is returned.

8-9. *Fibonacci Numbers*. The Fibonacci number sequence is 1, 1, 2, 3, 5, 8, 13, 21, etc. In other words, the next value of the sequence is the sum of the previous two values in the sequence. Write a routine that, given N, displays the value of the Nth Fibonacci number. For example, the first Fibonacci number is 1, the 6th is 8, and so on.

8-10. *Text Processing*. Determine the total number of vowels, consonants, and words (separated by spaces) in a text sentence. Ignore special cases for vowels and consonants such as "h," "y," "qu," etc. Extra credit: create code to handle those special case.

8-11. *Text Processing.* Write a program to ask the user to input a list of names, in the format "Last Name, First Name," i.e., last name, comma, first name. Write a function that manages the input so that when/if the user types the names in the wrong order, i.e., "First Name Last Name," the error is corrected, and the user is notified. This function should also keep track of the number of input mistakes. When the user is done, sort the list, and display the sorted names in "Last Name, First Name" order.

EXAMPLE input and output (you don't have to do it this way exactly):

```
% nametrack.py
Enter total number of names: 5

Please enter name 0: Smith, Joe
Please enter name 1: Mary Wong
>> Wrong format... should be Last, First.
>> You have done this 1 time(s) already. Fixing input. . .
Please enter name 2: Hamilton, Gerald
Please enter name 3: Royce, Linda
Please enter name 4: Winston Salem
>> Wrong format... should be Last, First.
>> You have done this 2 time(s) already. Fixing input. . .

The sorted list (by last name) is:
    Hamilton, Gerald
    Royce, Linda
    Salem, Winston
    Smith, Joe
    Wong, Mary
```

8-12. *(Integer) Bit Operators*. Write a program that takes begin and end values and prints out a decimal, binary, octal, hexadecimal chart like the one shown below. If any of the characters are printable ASCII characters, then print those, too. If none is, you may omit the ASCII column header.

```
SAMPLE OUTPUT 1
---------------
Enter begin value: 9
Enter end value: 18
DEC     BIN       OCT       HEX
-------------------------------
9       01001     11         9
10      01010     12         a
11      01011     13         b
12      01100     14         c
13      01101     15         d
14      01110     16         e
15      01111     17         f
16      10000     20         10
17      10001     21         11
18      10010     22         12
```

```
SAMPLE OUTPUT 2
---------------
Enter begin value: 26
Enter end value: 41
DEC     BIN        OCT     HEX     ASCII
----------------------------------------
26      011010     32       1a
27      011011     33       1b
28      011100     34       1c
29      011101     35       1d
30      011110     36       1e
31      011111     37       1f
32      100000     40       20
33      100001     41       21      !
34      100010     42       22      '
35      100011     43       23      #
36      100100     44       24      $
37      100101     45       25      %
38      100110     46       26      &
39      100111     47       27      '
40      101000     50       28      (
41      101001     51       29      )
```

8-13. *Performance.* In Section 8.5.2, we examined two basic ways of iterating over a sequence: (1) by sequence item, and (2) via sequence index. We pointed out at the end that the latter does not perform as well over the long haul (on my system here, a test suite shows performance is nearly twice as bad [83% worse]). Why do you think that is?