

The next major type on our built-in object tour is the Python *string*—an ordered collection of characters used to store and represent text-based information. We looked briefly at strings in Chapter 4. Here, we will revisit them in more depth, filling in some of the details we skipped then.

From a functional perspective, strings can be used to represent just about anything that can be encoded as text: symbols and words (e.g., your name), contents of text files loaded into memory, Internet addresses, Python programs, and so on.

You may have used strings in other languages, too. Python’s strings serve the same role as character arrays in languages such as C, but they are a somewhat higher-level tool than arrays. Unlike in C, in Python, strings come with a powerful set of processing tools. Also, unlike languages such as C, Python has no special type for single characters (like C’s `char`); instead, you just use one-character strings.

Strictly speaking, Python strings are categorized as immutable sequences, meaning that the characters they contain have a left-to-right positional order, and that they cannot be changed in-place. In fact, strings are the first representative of the larger class of objects called sequences that we will study here. Pay special attention to the sequence operations introduced in this chapter, because they will work the same on other sequence types we’ll explore later, such as lists and tuples.

Table 7-1 previews common string literals and operations we will discuss in this chapter. Empty strings are written as a pair of quotation marks (single or double) with nothing in between, and there are a variety of ways to code strings. For processing, strings support *expression* operations such as concatenation (combining strings), slicing (extracting sections), indexing (fetching by offset), and so on. Besides expressions, Python also provides a set of string *methods* that implement common string-specific tasks, as well as *modules* for more advanced text-processing tasks such as pattern matching. We’ll explore all of these later in the chapter.

Table 7-1. Common string literals and operations

Operation	Interpretation
<code>s1 = ''</code>	Empty string
<code>s2 = "spam's"</code>	Double quotes
<code>block = """..."""</code>	Triple-quoted blocks
<code>s3 = r'\temp\spam'</code>	Raw strings
<code>s4 = u'spam'</code>	Unicode strings
<code>s1 + s2</code> <code>s2 * 3</code>	Concatenate, repeat
<code>s2[i]</code> <code>s2[i:j]</code> <code>len(s2)</code>	Index, slice, length
<code>"a %s parrot" % type</code> <code>s2.find('pa')</code> <code>s2.rstrip()</code> <code>s2.replace('pa', 'xx')</code> <code>s1.split(',')</code> <code>s1.isdigit()</code> <code>s1.lower()</code>	String formatting String method calls: search, remove whitespace, replacement, split on delimiter, content test, case conversion, etc.
<code>for x in s2</code> <code>'spam' in s2</code>	Iteration, membership

Beyond the core set of string tools, Python also supports more advanced pattern-based string processing with the standard library's `re` (regular expression) module, introduced in Chapter 4. This chapter starts with an overview of string literal forms and basic string expressions, then looks at more advanced tools such as string methods and formatting.

## String Literals

By and large, strings are fairly easy to use in Python. Perhaps the most complicated thing about them is that there are so many ways to write them in your code:

- Single quotes: `'spa"m'`
- Double quotes: `"spa'm"`
- Triple quotes: `'''... spam ...'''`, `"""... spam ..."""`
- Escape sequences: `"s\tp\na\0m"`
- Raw strings: `r"C:\new\test.spm"`
- Unicode strings: `u'eggs\u0020spam'`

The single- and double-quoted forms are by far the most common; the others serve specialized roles. Let's take a quick look at each of these options.

## Single- and Double-Quoted Strings Are the Same

Around Python strings, single and double quote characters are interchangeable. That is, string literals can be written enclosed in either two single or two double quotes—the two forms work the same and return the same type of object. For example, the following two strings are identical, once coded:

```
➡ >>> 'shrubbery', "shrubbery"  
('shrubbery', 'shrubbery')
```

The reason for including both is that it allows you to embed a quote character of the other variety inside a string without escaping it with a backslash. You may embed a single quote character in a string enclosed in double quote characters, and vice versa:

```
➡ >>> 'knight"s', "knight's"  
('knight"s', "knight's")
```

Incidentally, Python automatically concatenates adjacent string literals in any expression, although it is almost as simple to add a + operator between them to invoke concatenation explicitly:

```
➡ >>> title = "Meaning " "of" " Life"      # Implicit concatenation  
>>> title  
'Meaning of Life'
```

Notice that adding commas between these strings would make a tuple, not a string. Also, notice in all of these outputs that Python prefers to print strings in single quotes, unless they embed one. You can also embed quotes by escaping them with backslashes:

```
➡ >>> 'knight\'s', "knight's"  
("knight's", 'knight's')
```

To understand why, you need to know how escapes work in general.

## Escape Sequences Represent Special Bytes

The last example embedded a quote inside a string by preceding it with a backslash. This is representative of a general pattern in strings: backslashes are used to introduce special byte codings known as escape sequences.

Escape sequences let us embed byte codes in strings that cannot easily be typed on a keyboard. The character \, and one or more characters following it in the string literal, are replaced with a single character in the resulting string object, which has the binary value specified by the escape sequence. For example, here is a five-character string that embeds a newline and a tab:

```
➡ >>> s = 'a\nb\tc'
```

The two characters `\n` stand for a single character—the byte containing the binary value of the newline character in your character set (usually, ASCII code 10). Similarly, the sequence `\t` is replaced with the tab character. The way this string looks when printed depends on how you print it. The interactive echo shows the special characters as escapes, but `print` interprets them instead:

```
>>> s
'a\nb\tc'
>>> print s
a
b      c
```

To be completely sure how many bytes are in this string, use the built-in `len` function—it returns the actual number of bytes in a string, regardless of how it is displayed:

```
>>> len(s)
5
```

This string is five bytes long: it contains an ASCII *a* byte, a newline byte, an ASCII *b* byte, and so on. Note that the original backslash characters are not really stored with the string in memory. For coding such special bytes, Python recognizes a full set of escape code sequences, listed in Table 7-2.

Table 7-2. String backslash characters

Escape	Meaning
<code>\newline</code>	Ignored (continuation)
<code>\\</code>	Backslash (keeps a <code>\</code> )
<code>\'</code>	Single quote (keeps <code>'</code> )
<code>\"</code>	Double quote (keeps <code>"</code> )
<code>\a</code>	Bell
<code>\b</code>	Backspace
<code>\f</code>	Formfeed
<code>\n</code>	Newline (linefeed)
<code>\r</code>	Carriage return
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\N{id}</code>	Unicode database ID
<code>\uhhhh</code>	Unicode 16-bit hex
<code>\Uhhhh...</code>	Unicode 32-bit hex <sup>a</sup>
<code>\xhh</code>	Hex digits value
<code>\ooo</code>	Octal digits value
<code>\0</code>	Null (doesn't end string)
<code>\other</code>	Not an escape (kept)

<sup>a</sup> The `\Uhhhh...` escape sequence takes exactly eight hexadecimal digits (h); both `\u` and `\U` can be used only in Unicode string literals.

Some escape sequences allow you to embed absolute binary values into the bytes of a string. For instance, here's a five-character string that embeds two binary zero bytes:

```
>>> s = 'a\0b\0c'
>>> s
'a\x00b\x00c'
>>> len(s)
5
```

In Python, the zero (null) byte does not terminate a string the way it typically does in C. Instead, Python keeps both the string's length and text in memory. In fact, no character terminates a string in Python. Here's a string that is all absolute binary escape codes—a binary 1 and 2 (coded in octal), followed by a binary 3 (coded in hexadecimal):

```
>>> s = '\001\002\x03'
>>> s
'\x01\x02\x03'
>>> len(s)
3
```

This becomes more important to know when you process binary data files in Python. Because their contents are represented as strings in your scripts, it's okay to process binary files that contain any sorts of binary byte values (more on files in Chapter 9).\*

Finally, as the last entry in Table 7-2 implies, if Python does not recognize the character after a \ as being a valid escape code, it simply keeps the backslash in the resulting string:

```
>>> x = "C:\py\code"          # Keeps \ literally
>>> x
'C:\\py\\code'
>>> len(x)
10
```

Unless you're able to commit all of Table 7-2 to memory, though, you probably shouldn't rely on this behavior.† To code literal backslashes explicitly such that they are retained in your strings, double them up (\\ is an escape for \) or use raw strings, described in the next section.

## Raw Strings Suppress Escapes

As we've seen, escape sequences are handy for embedding special byte codes within strings. Sometimes, though, the special treatment of backslashes for introducing

\* If you're especially interested in binary data files, the chief distinction is that you open them in binary mode (using open mode flags with a b, such as 'rb', 'wb', and so on). See also the standard struct module introduced in Chapter 9, which can parse binary data loaded from a file.

† In classes, I've met people who have indeed committed most or all of this table to memory; I'd normally think that's really sick, but for the fact that I'm a member of the set, too.

escapes can lead to trouble. It's surprisingly common, for instance, to see Python newcomers in classes trying to open a file with a filename argument that looks something like this:

```
myfile = open('C:\new\text.dat', 'w')
```

thinking that they will open a file called *text.dat* in the directory *C:\new*. The problem here is that `\n` is taken to stand for a newline character, and `\t` is replaced with a tab. In effect, the call tries to open a file named *C:(newline)ew(tab)ext.dat*, with usually less than stellar results.

This is just the sort of thing that raw strings are useful for. If the letter *r* (uppercase or lowercase) appears just before the opening quote of a string, it turns off the escape mechanism. The result is that Python retains your backslashes literally, exactly as you type them. Therefore, to fix the filename problem, just remember to add the letter *r* on Windows:

```
myfile = open(r'C:\new\text.dat', 'w')
```

Alternatively, because two backslashes are really an escape sequence for one backslash, you can keep your backslashes by simply doubling them up:

```
myfile = open('C:\\new\\text.dat', 'w')
```

In fact, Python itself sometimes uses this doubling scheme when it prints strings with embedded backslashes:


```
>>> path = r'C:\new\text.dat'
>>> path                                     # Show as Python code
'C:\\new\\text.dat'
>>> print path                             # User-friendly format
C:\new\text.dat
>>> len(path)                             # String length
15
```

As with numeric representation, the default format at the interactive prompt prints results as if they were code, and therefore escapes backslashes in the output. The `print` statement provides a more user-friendly format that shows that there is actually only one backslash in each spot. To verify this is the case, you can check the result of the built-in `len` function, which returns the number of bytes in the string, independent of display formats. If you count the characters in the `print path` output, you'll see that there really is just one character per backslash, for a total of 15.

Besides directory paths on Windows, raw strings are also commonly used for regular expressions (text pattern matching, supported with the `re` module introduced in Chapter 4). Also note that Python scripts can usually use *forward* slashes in directory paths on Windows and Unix because Python tries to interpret paths portably. Raw strings are useful if you code paths using native Windows backslashes, though.

## Triple Quotes Code Multiline Block Strings

So far, you've seen single quotes, double quotes, escapes, and raw strings in action. Python also has a triple-quoted string literal format, sometimes called a *block string*, that is a syntactic convenience for coding multiline text data. This form begins with three quotes (of either the single or double variety), is followed by any number of lines of text, and is closed with the same triple-quote sequence that opened it. Single and double quotes embedded in the string's text may be, but do not have to be, escaped—the string does not end until Python sees three unescaped quotes of the same kind used to start the literal. For example:




```
>>> mantra = """Always look
...   on the bright
...   side of life."""
>>>
>>> mantra
'Always look\n on the bright\nside of life.'
```

This string spans three lines (in some interfaces, the interactive prompt changes to ... on continuation lines; IDLE simply drops down one line). Python collects all the triple-quoted text into a single multiline string, with embedded newline characters (\n) at the places where your code has line breaks. Notice that, as in the literal, the second line in the result has a leading space, but the third does not—what you type is truly what you get.

Triple-quoted strings are useful any time you need multiline text in your program; for example, to embed multiline error messages or HTML or XML code in your source code files. You can embed such blocks directly in your scripts without resorting to external text files, or explicit concatenation and newline characters.

Triple-quoted strings are also commonly used for documentation strings, which are string literals that are taken as comments when they appear at specific points in your file (more on these later in the book). These don't have to be triple-quoted blocks, but they usually are to allow for multiline comments.

Finally, triple-quoted strings are also often used as a horribly hackish way to temporarily disable lines of code during development (OK, it's not really too horrible, and it's actually a fairly common practice). If you wish to turn off a few lines of code and run your script again, simply put three quotes above and below them, like this:



```
X = 1
"""
import os
print os.getcwd()
"""
Y = 2
```


I said this was hackish because Python really does make a string out of the lines of code disabled this way, but this is probably not significant in terms of performance. For large sections of code, it's also easier than manually adding hash marks before each line and later removing them. This is especially true if you are using a text editor that does not have support for editing Python code specifically. In Python, practicality often beats aesthetics.

## Unicode Strings Encode Larger Character Sets

The last way to write strings in your scripts is perhaps the most specialized, and it's rarely observed outside of web and XML processing. *Unicode strings* are sometimes called “wide” character strings. Because each character may be represented with more than one byte in memory, Unicode strings allow programs to encode richer character sets than standard strings.

Unicode strings are typically used to support *internationalization* of applications (sometimes referred to as “i18n,” to compress the 18 characters between the first and last characters of the term). For instance, they allow programmers to directly support European or Asian character sets in Python scripts. Because such character sets have more characters than can be represented by single bytes, Unicode is normally used to process these forms of text.

In Python, you can code Unicode strings in your scripts by adding the letter *U* (lower- or uppercase) just before the opening quote:



```
>>> u'spam'
u'spam'
```


Technically, this syntax generates a Unicode string object, which is a different data type from the normal string type. However, Python allows you to freely mix Unicode and normal strings in expressions, and converts up to Unicode for mixed-type results (more on + concatenation in the next section):



```
>>> 'ni' + u'spam'      # Mixed string types
u'nispam'
```

In fact, Unicode strings are defined to support all the usual string-processing operations you'll meet in the next section, so the difference in types is often trivial to your code. Like normal strings, Unicode strings may be concatenated, indexed, sliced, matched with the *re* module, and so on, and cannot be changed in-place.


If you ever need to convert between the two types explicitly, you can use the built-in *str* and *unicode* functions:



```
>>> str(u'spam')        # Unicode to normal
'spam'
>>> unicode('spam')     # Normal to Unicode
u'spam'
```



Because Unicode is designed to handle multibyte characters, you can also use the special `\u` and `\U` escapes to encode binary character values that are larger than 8 bits:



```
>>> u'ab\x20cd'           # 8-bit/1-byte characters
u'ab cd'
>>> u'ab\u0020cd'         # 2-byte characters
u'ab cd'
>>> u'ab\U00000020cd'     # 4-byte characters
u'ab cd'
```

The first of these statements embeds the binary code for a space character; its binary value in hexadecimal notation is `x20`. The second and third statements do the same, but give the value in 2-byte and 4-byte Unicode escape notation.

Even if you don't think you will need Unicode strings, you might use them without knowing it. Because some programming interfaces (e.g., the COM API on Windows, and some XML parsers) represent text as Unicode, it may find its way into your scripts as API inputs or results, and you may sometimes need to convert back and forth between normal and Unicode types.

Because Python treats the two string types interchangeably in most contexts, though, the presence of Unicode strings is often transparent to your code—you can largely ignore the fact that text is being passed around as Unicode objects and use normal string operations.

Unicode is a useful addition to Python, and because support is built-in, it's easy to handle such data in your scripts when needed. Of course, there's a lot more to say about the Unicode story. For example:

- Unicode objects provide an `encode` method that converts a Unicode string into a normal 8-bit string using a specific encoding.
- The built-in function `unicode` and module `codecs` support registered Unicode “codecs” (for “COders and DECode rs”).
- The `open` function within the `codecs` module allows for processing Unicode text files, where each character is stored as more than one byte.
- The `unicodedata` module provides access to the Unicode character database.
- The `sys` module includes calls for fetching and setting the default Unicode encoding scheme (the default is usually ASCII).
- You may combine the `raw` and Unicode string formats (e.g., `ur'a\b\c'`).

Because Unicode is a relatively advanced and not so commonly used tool, we won't discuss it further in this introductory text. See the Python standard manual for the rest of the Unicode story.



In Python 3.0, the string type will mutate somewhat: the current `str` string type will always be Unicode in 3.0, and there will be a new “bytes” type that will be a mutable sequence of small integers useful for representing short character strings. Some file read operations may return bytes instead of `str` (reading binary files, for example). This is still on the drawing board, so consult 3.0 release notes for details.

## Strings in Action

Once you’ve created a string with the literal expressions we just met, you will almost certainly want to do things with it. This section and the next two demonstrate string basics, formatting, and methods—the first line of text-processing tools in the Python language.

### Basic Operations

Let’s begin by interacting with the Python interpreter to illustrate the basic string operations listed in Table 7-1. Strings can be concatenated using the `+` operator and repeated using the `*` operator:

```
% python
>>> len('abc')           # Length: number of items
3
>>> 'abc' + 'def'        # Concatenation: a new string
'abcdef'
>>> 'Ni!' * 4             # Repetition: like "Ni!" + "Ni!" + ...
'Ni!Ni!Ni!Ni!'
```

Formally, adding two string objects creates a new string object, with the contents of its operands joined. Repetition is like adding a string to itself a number of times. In both cases, Python lets you create arbitrarily sized strings; there’s no need to predeclare anything in Python, including the sizes of data structures.\* The `len` built-in function returns the length of a string (or any other object with a length).


Repetition may seem a bit obscure at first, but it comes in handy in a surprising number of contexts. For example, to print a line of 80 dashes, you can count up to 80, or let Python count for you:

```
>>> print '----- ...more... ---'    # 80 dashes, the hard way
>>> print '-'*80                       # 80 dashes, the easy way
```

\* Unlike C character arrays, you don’t need to allocate or manage storage arrays when using Python strings; simply create string objects as needed, and let Python manage the underlying memory space. As discussed in the prior chapter, Python reclaims unused objects’ memory space automatically, using a reference-count garbage-collection strategy. Each object keeps track of the number of names, data structures, etc., that reference it; when the count reaches zero, Python frees the object’s space. This scheme means Python doesn’t have to stop and scan all the memory to find unused space to free (an additional garbage component also collects cyclic objects).

Notice that operator overloading is at work here already: we're using the same + and \* operators that perform addition and multiplication when using numbers. Python does the correct operation because it knows the types of the objects being added and multiplied. But be careful: the rules aren't quite as liberal as you might expect. For instance, Python doesn't allow you to mix numbers and strings in + expressions: 'abc'+9 raises an error instead of automatically converting 9 to a string.

As shown in the last line in Table 7-1, you can also iterate over strings in loops using for statements, and test membership with the in expression operator, which is essentially a search:




```
>>> myjob = "hacker"
>>> for c in myjob: print c,          # Step through items
...
h a c k e r
>>> "k" in myjob                     # Found
True
>>> "z" in myjob                     # Not found
False
```

The for loop assigns a variable to successive items in a sequence (here, a string), and executes one or more statements for each item. In effect, the variable c becomes a cursor stepping across the string here. We will discuss iteration tools like these in more detail later in this book.

## Indexing and Slicing

Because strings are defined as ordered collections of characters, we can access their components by position. In Python, characters in a string are fetched by *indexing*—providing the numeric offset of the desired component in square brackets after the string. You get back the one-character string at the specified position.

As in the C language, Python offsets start at 0, and end at one less than the length of the string. Unlike C, however, Python also lets you fetch items from sequences such as strings using *negative* offsets. Technically, a negative offset is added to the length of a string to derive a positive offset. You can also think of negative offsets as counting backward from the end. The following interaction demonstrates:



```
>>> S = 'spam'
>>> S[0], S[-2]                     # Indexing from front or end
('s', 'a')
>>> S[1:3], S[1:], S[:-1]           # Slicing: extract a section
('pa', 'pam', 'spa')
```

The first line defines a four-character string, and assigns it the name `S`. The next line indexes it in two ways: `S[0]` fetches the item at offset 0 from the left (the one-character string `'s'`), and `S[-2]` gets the item at offset 2 from the end (or equivalently, at offset  $(4 + -2)$  from the front). Offsets and slices map to cells as shown in Figure 7-1.\*

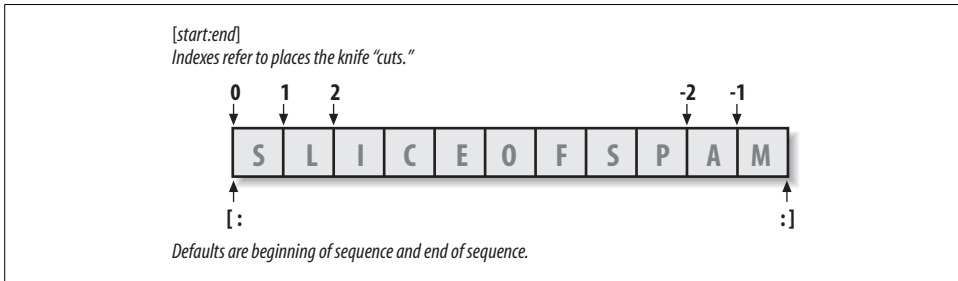


Figure 7-1. Offsets and slices: positive offsets start from the left end (offset 0 is the first item), and negatives count back from the right end (offset `-1` is the last item). Either kind of offset can be used to give positions in indexing and slicing.

The last line in the preceding example demonstrates *slicing*. Probably the best way to think of slicing is that it is a form of *parsing* (analyzing structure), especially when applied to strings—it allows us to extract an entire section (substring) in a single step. Slices can be used to extract columns of data, chop off leading and trailing text, and more. We’ll look at another slicing-as-parsing example later in this chapter.

Here’s how slicing works. When you index a sequence object such as a string on a pair of offsets separated by a colon, Python returns a new object containing the contiguous section identified by the offset pair. The left offset is taken to be the lower bound (inclusive), and the right is the upper bound (noninclusive). Python fetches all items from the lower bound up to but not including the upper bound, and returns a new object containing the fetched items. If omitted, the left and right bounds default to 0, and the length of the object you are slicing, respectively.

For instance, in the example we just looked at, `S[1:3]` extracts the items at offsets 1 and 2. That is, it grabs the second and third items, and stops before the fourth item at offset 3. Next `S[1:]` gets *all items beyond the first*—the upper bound, which is not specified, defaults to the length of the string. Finally, `S[:-1]` fetches *all but the last item*—the lower bound defaults to 0, and `-1` refers to the last item, noninclusive.

This may seem confusing at first glance, but indexing and slicing are simple and powerful tools to use, once you get the knack. Remember, if you’re unsure about what a slice means, try it out interactively. In the next chapter, you’ll see that it’s also possible to change an entire section of a certain object in one step by assigning to a slice. Here’s a summary of the details for reference:

\* More mathematically minded readers (and students in my classes) sometimes detect a small asymmetry here: the leftmost item is at offset 0, but the rightmost is at offset `-1`. Alas, there is no such thing as a distinct `-0` value in Python.

- Indexing (`S[i]`) fetches components at offsets:
  - The first item is at offset 0.
  - Negative indexes mean to count backward from the end or right.
  - `S[0]` fetches the first item.
  - `S[-2]` fetches the second item from the end (like `S[len(S)-2]`).
- Slicing (`S[i:j]`) extracts contiguous sections of a sequence:
  - The upper bound is noninclusive.
  - Slice boundaries default to 0 and the sequence length, if omitted.
  - `S[1:3]` fetches items at offsets 1 up to, but not including, 3.
  - `S[1:]` fetches items at offset 1 through the end (length).
  - `S[:3]` fetches items at offset 0 up to, but not including, 3.
  - `S[:-1]` fetches items at offset 0 up to, but not including, the last item.
  - `S[:]` fetches items at offsets 0 through the end—this effectively performs a top-level copy of `S`.

The last item listed here turns out to be a very common trick: it makes a full top-level *copy* of a sequence object—an object with the same value, but a distinct piece of memory (you’ll find more on copies in Chapter 9). This isn’t very useful for immutable objects like strings, but it comes in handy for objects that may be changed in-place, such as lists. In the next chapter, you’ll also see that the syntax used to index by offset (square brackets) is used to index dictionaries by key as well; the operations look the same, but have different interpretations.

### Extended slicing: the third limit

In Python 2.3, slice expressions grew support for an optional third index, used as a *step* (sometimes called a *stride*). The step is added to the index of each item extracted. The full-blown form of a slice is now `X[I:J:K]`, which means “extract all the items in `X`, from offset `I` through `J-1`, by `K`.” The third limit, `K`, defaults to 1, which is why normally all items in a slice are extracted from left to right. If you specify an explicit value, however, you can use the third limit to skip items or to reverse their order.

For instance, `X[1:10:2]` will fetch *every other item* in `X` from offsets 1–9; that is, it will collect the items at offsets 1, 3, 5, 7, and 9. As usual, the first and second limits default to 0, and the length of the sequence, respectively, so `X[::2]` gets every other item from the beginning to the end of the sequence:

```

>>> S = 'abcdefghijklmnop'
>>> S[1:10:2]
'bdfhj'
>>> S[::2]
'acegikmo'

```

You can also use a negative stride. For example, the slicing expression `"hello"[::-1]` returns the new string `"olleh"`—the first two bounds default to 0, and the length of the sequence, as before, and a stride of `-1` indicates that the slice should go from right to left instead of the usual left to right. The effect, therefore, is to *reverse* the sequence:

```
>>> S = 'hello'
>>> S[::-1]
'olleh'
```

With a negative stride, the meanings of the first two bounds are essentially reversed. That is, the slice `S[5:1:-1]` fetches the items from 2 to 5, in reverse order (the result contains items from offsets 5, 4, 3, and 2):

```
>>> S = 'abcdefg'
>>> S[5:1:-1]
'fdec'
```

Skipping and reversing like this are the most common use cases for three-limit slices, but see Python's standard library manual for more details, or run a few experiments interactively—there is more to the story than we will cover here. We'll revisit three-limit slices again later in this book, in conjunction with the `for` loop statement.

## String Conversion Tools

One of Python's design mottos is that it refuses the temptation to guess. As a prime example, you cannot add a number and a string together in Python, even if the string looks like a number (i.e., is all digits):

```
>>> "42" + 1
TypeError: cannot concatenate 'str' and 'int' objects
```

This is by design: because `+` can mean both addition and concatenation, the choice of conversion would be ambiguous. So, Python treats this as an error. In Python, magic is generally omitted if it will make your life more complex.

What to do, then, if your script obtains a number as a text string from a file or user interface? The trick is that you need to employ conversion tools before you can treat a string like a number, or vice versa. For instance:


```
>>> int("42"), str(42)           # Convert from/to string
(42, '42')
>>> repr(42), `42`              # Convert to as-code string
('42', '42')
```

The `int` function converts a string to a number, and the `str` function converts a number to its string representation (essentially, what it looks like when printed). The `repr` function and its older equivalent, the backquotes expression, also convert an object to its string representation, but these return the object as a string of code that can be rerun to recreate the object (for strings, the result has quotes around it if displayed

## Why You Will Care: Slices

Throughout this book, I will include common use case sidebars (such as this one) to give you a peek at how some of the language features being introduced are typically used in real programs. Because you won't be able to make much sense of real use cases until you've seen most of the Python picture, these sidebars necessarily contain many references to topics not introduced yet; at most, you should consider them previews of ways that you may find these abstract language concepts useful for common programming tasks.

For instance, you'll see later that the argument words listed on a system command line used to launch a Python program are made available in the `argv` attribute of the built-in `sys` module:



```
# File echo.py
import sys
print sys.argv

% python echo.py -a -b -c
['echo.py', '-a', '-b', '-c']
```


Usually, you're only interested in inspecting the arguments that follow the program name. This leads to a very typical application of slices: a single slice expression can be used to return all but the first item of a list. Here, `sys.argv[1:]` returns the desired list, `['-a', '-b', '-c']`. You can then process this list without having to accommodate the program name at the front.

Slices are also often used to clean up lines read from input files. If you know that a line will have an end-of-line character at the end (a `\n` newline marker), you can get rid of it with a single expression such as `line[:-1]`, which extracts all but the last character in the line (the lower limit defaults to 0). In both cases, slices do the job of logic that must be explicit in a lower-level language.

Note that calling the `line.rstrip` method is often preferred for stripping newline characters because this call leaves the line intact if it has no newline character at the end—a common case for files created with some text-editing tools. Slicing works if you're sure the line is properly terminated.

with the `print` statement). See the “str and repr Display Formats” sidebar in Chapter 5 on the difference between `str` and `repr` for more on this topic. Of these, `int` and `str` are the generally prescribed conversion techniques.

Although you can't mix strings and number types around operators such as `+`, you can manually convert operands before that operation if needed:



```
>>> S = "42"
>>> I = 1
>>> S + I
TypeError: cannot concatenate 'str' and 'int' objects
```

```
>>> int(5) + I                # Force addition
43

>>> S + str(I)                # Force concatenation
'421'
```

Similar built-in functions handle floating-point number conversions to and from strings:

```
>>> str(3.1415), float("1.5")
('3.1415', 1.5)

>>> text = "1.234E-10"
>>> float(text)
1.2340000000000001e-010
```

Later, we'll further study the built-in `eval` function; it runs a string containing Python expression code, and so can convert a string to any kind of object. The functions `int` and `float` convert only to numbers, but this restriction means they are usually faster (and more secure, because they do not accept arbitrary expression code). As we saw in Chapter 5, the string formatting expression also provides a way to convert numbers to strings. We'll discuss formatting further later in this chapter.

## Character code conversions

On the subject of conversions, it is also possible to convert a single character to its underlying ASCII integer code by passing it to the built-in `ord` function—this returns the actual binary value of the corresponding byte in memory. The `chr` function performs the inverse operation, taking an ASCII integer code and converting it to the corresponding character:

```
>>> ord('s')
115
>>> chr(115)
's'
```

You can use a loop to apply these functions to all characters in a string. These tools can also be used to perform a sort of string-based math. To advance to the next character, for example, convert and do the math in integer:

```
>>> S = '5'
>>> S = chr(ord(S) + 1)
>>> S
'6'
>>> S = chr(ord(S) + 1)
>>> S
'7'
```

At least for single-character strings, this provides an alternative to using the built-in `int` function to convert from string to integer:

```
>>> int('5')
5
>>> ord('5') - ord('0')
5
```



Such conversions can be used in conjunction with a looping statement to convert a string of binary digits to their corresponding integer values—each time through, multiply the current value by 2, and add the next digit’s integer value:

```
>>> B = '1101'
>>> I = 0
>>> while B:
...     I = I * 2 + (ord(B[0]) - ord('0'))
...     B = B[1:]
...
>>> I
13
```

A left-shift operation ( $I \ll 1$ ) would have the same effect as multiplying by 2 here. Because we haven’t studied loops in detail yet, though, we’ll leave implementing that as a suggested experiment.

## Changing Strings

Remember the term “immutable sequence”? The immutable part means that you can’t change a string in-place (e.g., by assigning to an index):

```
>>> S = 'spam'
>>> S[0] = "x"
Raises an error!
```

So how do you modify text information in Python? To change a string, you need to build and assign a new string using tools such as concatenation and slicing, and then, if desired, assign the result back to the string’s original name:

```
>>> S = S + 'SPAM!'           # To change a string, make a new one
>>> S
'spamSPAM!'
>>> S = S[:4] + 'Burger' + S[-1]
>>> S
'spamBurger!'
```


The first example adds a substring at the end of `S`, by concatenation; really, it makes a new string and assigns it back to `S`, but you can think of this as “changing” the original string. The second example replaces four characters with six by slicing, indexing, and concatenating. As you’ll see later in this chapter, you can achieve similar effects with string method calls like `replace`. Here’s a sneak peek:

```
>>> S = 'sploit'
>>> S = S.replace('pl', 'pamal')
>>> S
'spamalot'
```

Like every operation that yields a new string value, string methods generate new string objects. If you want to retain those objects, you can assign it to a variable name. Generating a new string object for each string change is not as inefficient as it may sound—remember, as discussed in the preceding chapter, Python automatically

garbage collects (reclaims the space of) old unused string objects as you go, so newer objects reuse the space held by prior values. Python is usually more efficient than you might expect.

Finally, it's also possible to build up new text values with string formatting expressions:



```
>>> 'That is %d %s bird!' % (1, 'dead')    # Like C sprintf
That is 1 dead bird!
```

This turns out to be a powerful operation. The next section shows how it works.

## String Formatting


Python defines the `%` binary operator to work on strings (you may recall that this is also the remainder of division, or modulus, operator for numbers). When applied to strings, this operator serves the same role as C's `sprintf` function; the `%` provides a simple way to format values as strings, according to a format definition string. In short, the `%` operator provides a compact way to code multiple string substitutions.

To format strings:

1. On the left of the `%` operator, provide a format string containing one or more embedded conversion targets, each of which starts with a `%` (e.g., `%d`).
2. On the right of the `%` operator, provide the object (or objects, in parentheses) that you want Python to insert into the format string on the left in place of the conversion target (or targets).

For instance, in the last example, we looked at in the prior section, the integer `1` replaces the `%d` in the format string on the left, and the string `'dead'` replaces the `%s`. The result is a new string that reflects these two substitutions.

Technically speaking, string formatting expressions are usually optional—you can generally do similar work with multiple concatenations and conversions. However, formatting allows us to combine many steps into a single operation. It's powerful enough to warrant a few more examples:



```
>>> exclamation = "Ni"
>>> "The knights who say %s!" % exclamation
'The knights who say Ni!'

>>> "%d %s %d you" % (1, 'spam', 4)
'1 spam 4 you'

>>> "%s -- %s -- %s" % (42, 3.14159, [1, 2, 3])
'42 -- 3.14159 -- [1, 2, 3]'
```

The first example here plugs the string `"Ni"` into the target on the left, replacing the `%s` marker. In the second example, three values are inserted into the target string. Note that when you're inserting more than one value, you need to group the values on the right in parentheses (i.e., put them in a tuple).

The third example again inserts three values—an integer, a floating-point object, and a list object—but notice that all of the targets on the left are `%s`, which stands for conversion to string. As every type of object can be converted to a string (the one used when printing), every object type works with the `%s` conversion code. Because of this, unless you will be doing some special formatting, `%s` is often the only code you need to remember for the formatting expression.

Again, keep in mind that formatting always makes a new string, rather than changing the string on the left; because strings are immutable, it must work this way. As before, assign the result to a variable name if you need to retain it.

## Advanced String Formatting

For more advanced type-specific formatting, you can use any of the conversion codes listed in Table 7-3 in formatting expressions. C programmers will recognize most of these because Python string formatting supports all the usual C `printf` format codes (but returns the result, instead of displaying it, like `printf`). Some of the format codes in the table provide alternative ways to format the same type; for instance, `%e`, `%f`, and `%g` provide alternative ways to format floating-point numbers.

Table 7-3. String-formatting codes

Code	Meaning
<code>%s</code>	String (or any object)
<code>%r</code>	s, but uses <code>repr</code> , not <code>str</code>
<code>%c</code>	Character
<code>%d</code>	Decimal (integer)
<code>%i</code>	Integer
<code>%u</code>	Unsigned (integer)
<code>%o</code>	Octal integer
<code>%x</code>	Hex integer
<code>%X</code>	x, but prints uppercase
<code>%e</code>	Floating-point exponent
<code>%E</code>	e, but prints uppercase
<code>%f</code>	Floating-point decimal
<code>%g</code>	Floating-point e or f
<code>%G</code>	Floating-point E or f
<code>%%</code>	Literal %

In fact, conversion targets in the format string on the expression's left side support a variety of conversion operations with a fairly sophisticated syntax all their own. The general structure of conversion targets looks like this:

➡ `%[(name)][flags][width][.precision]code`

The character codes in Table 7-3 show up at the end of the target string. Between the % and the character code, you can do any of the following: provide a dictionary key; list flags that specify things like left justification (-), numeric sign (+), and zero fills (0); give a total field width and the number of digits after a decimal point; and more.

Formatting target syntax is documented in full in the Python standard manuals, but to demonstrate common usage, let's look at a few examples. This one formats integers by default, and then in a six-character field with left justification, and zero padding:

```
>>> x = 1234
>>> res = "integers: ...%d...%-6d...%06d" % (x, x, x)
>>> res
'integers: ...1234...1234 ...001234'
```

The %e, %f, and %g formats display floating-point numbers in different ways, as the following interaction demonstrates:

```
>>> x = 1.23456789
>>> x
1.2345678899999999

>>> '%e | %f | %g' % (x, x, x)
'1.234568e+000 | 1.234568 | 1.23457'
```

For floating-point numbers, you can achieve a variety of additional formatting effects by specifying left justification, zero padding, numeric signs, field width, and digits after the decimal point. For simpler tasks, you might get by with simply converting to strings with a format expression or the str built-in function shown earlier:

```
>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23 | 01.23 | +001.2'

>>> "%s" % x, str(x)
('1.23456789', '1.23456789')
```

## Dictionary-Based String Formatting

String formatting also allows conversion targets on the left to refer to the keys in a dictionary on the right to fetch the corresponding values. I haven't told you much about dictionaries yet, so here's an example that demonstrates the basics:

```
>>> "%(n)d %(x)s" % {"n":1, "x":"spam"}
'1 spam'
```

Here, the (n) and (x) in the format string refer to keys in the dictionary literal on the right, and fetch their associated values. Programs that generate text such as HTML or XML often use this technique—you can build up a dictionary of values, and substitute them all at once with a single formatting expression that uses key-based references:

```

>>> reply = """
Greetings...
Hello %(name)s!
Your age squared is %(age)s
"""

>>> values = {'name': 'Bob', 'age': 40}
>>> print reply % values

Greetings...
Hello Bob!
Your age squared is 40

```

This trick is also used in conjunction with the `vars` built-in function, which returns a dictionary containing all the variables that exist in the place it is called:

```

>>> food = 'spam'
>>> age = 40
>>> vars()
{'food': 'spam', 'age': 40, ...many more... }

```

When used on the right of a format operation, this allows the format string to refer to variables by name (i.e., by dictionary key):

```

>>> "%(age)d %(food)s" % vars()
'40 spam'

```

We'll study dictionaries in more depth in Chapter 8. See also Chapter 5 for examples that convert to hexadecimal and octal number strings with the `%x` and `%o` formatting target codes.

## String Methods

In addition to expression operators, strings provide a set of *methods* that implement more sophisticated text-processing tasks. Methods are simply functions that are associated with particular objects. Technically, they are attributes attached to objects that happen to reference callable functions. In Python, methods are specific to object types—string methods, for example, work only on string objects.

In finer-grained detail, functions are packages of code, and method calls combine two operations at once (an attribute fetch, and a call):

### *Attribute fetches*

An expression of the form *object.attribute* means “fetch the value of *attribute* in *object*.”

### *Call expressions*

An expression of the form *function(arguments)* means “invoke the code of *function*, passing zero or more comma-separated *argument* objects to it, and return *function*’s result value.”

Putting these two together allows us to call a method of an object. The method call expression `object.method(arguments)` is evaluated from left to right—that is, Python will first fetch the *method* of the *object*, and then call it, passing in the *arguments*. If the method computes a result, it will come back as the result of the entire method-call expression.

As you'll see throughout this part of the book, most objects have callable methods, and all are accessed using this same method-call syntax. To call an object method, you have to go through an existing object. Let's move on to some examples to see how.

## String Method Examples: Changing Strings

Table 7-4 summarizes the call patterns for built-in string methods (be sure to check Python's standard library manual for the most up-to-date list, or run a help call on any string interactively). String methods in this table implement higher-level operations such as splitting and joining, case conversions, content tests, and substring searches.

Table 7-4. String method calls

<code>S.capitalize()</code>	<code>S.ljust(width)</code>
<code>S.center(width)</code>	<code>S.lower()</code>
<code>S.count(sub [, start [, end]])</code>	<code>S.lstrip()</code>
<code>S.encode([encoding [,errors]])</code>	<code>S.replace(old, new [, maxsplit])</code>
<code>S.endswith(suffix [, start [, end]])</code>	<code>S.rfind(sub [,start [,end]])</code>
<code>S.expandtabs([tabsize])</code>	<code>S.rindex(sub [, start [, end]])</code>
<code>S.find(sub [, start [, end]])</code>	<code>S.rjust(width)</code>
<code>S.index(sub [, start [, end]])</code>	<code>S.rstrip()</code>
<code>S.isalnum()</code>	<code>S.split([sep [,maxsplit]])</code>
<code>S.isalpha()</code>	<code>S.splitlines([keepends])</code>
<code>S.isdigit()</code>	<code>S.startswith(prefix [, start [, end]])</code>
<code>S.islower()</code>	<code>S.strip()</code>
<code>S.isspace()</code>	<code>S.swapcase()</code>
<code>S.istitle()</code>	<code>S.title()</code>
<code>S.isupper()</code>	<code>S.translate(table [, delchars])</code>
<code>S.join(seq)</code>	<code>S.upper()</code>

Now, let's work through some code that demonstrates some of the most commonly used methods in action, and illustrates Python text-processing basics along the way. As we've seen, because strings are immutable, they cannot be changed in-place directly. To make a new text value from an existing string, you construct a new string with operations such as slicing and concatenation. For example, to replace two characters in the middle of a string, you can use code like this:

```

>>> S = 'spammy'
>>> S = S[:3] + 'xx' + S[5:]
>>> S
'spaxxy'

```

But, if you're really just out to replace a substring, you can use the string `replace` method instead:

```

>>> S = 'spammy'
>>> S = S.replace('mm', 'xx')
>>> S
'spaxxy'

```

The `replace` method is more general than this code implies. It takes as arguments the original substring (of any length), and the string (of any length) to replace it with, and performs a global search and replace:

```

>>> 'aa$bb$cc$dd'.replace('$', 'SPAM')
'aaSPAMbbSPAMccSPAMdd'

```

In such a role, `replace` can be used as a tool to implement template replacements (e.g., in form letters). Notice that this time we simply printed the result, instead of assigning it to a name—you need to assign results to names only if you want to retain them for later use.

If you need to replace one fixed-size string that can occur at any offset, you can do a replacement again, or search for the substring with the string `find` method and then slice:

```

>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> where = S.find('SPAM')           # Search for position
>>> where                                # Occurs at offset 4
4
>>> S = S[:where] + 'EGGS' + S[(where+4):]
>>> S
'xxxxEGGSxxxxSPAMxxxx'

```

The `find` method returns the offset where the substring appears (by default, searching from the front), or `-1` if it is not found. Another option is to use `replace` with a third argument to limit it to a single substitution:

```

>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> S.replace('SPAM', 'EGGS')        # Replace all
'xxxxEGGSxxxxEGGSxxxx'

>>> S.replace('SPAM', 'EGGS', 1)     # Replace one
'xxxxEGGSxxxxSPAMxxxx'

```

Notice that `replace` returns a new string object each time. Because strings are immutable, methods never really change the subject strings in-place, even if they are called “replace”!

The fact that concatenation operations and the `replace` method generate new string objects each time they are run is actually a potential downside of using them to

change strings. If you have to apply many changes to a very large string, you might be able to improve your script's performance by converting the string to an object that does support in-place changes:

```
>>> S = 'spammy'
>>> L = list(S)
>>> L
['s', 'p', 'a', 'm', 'm', 'y']
```

The built-in `list` function (or an object construction call) builds a new list out of the items in any sequence—in this case, “exploding” the characters of a string into a list. Once the string is in this form, you can make multiple changes to it without generating a new copy for each change:

```
>>> L[3] = 'x'
>>> L[4] = 'x'
>>> L
['s', 'p', 'a', 'x', 'x', 'y']
```

*# Works for lists, not strings*

If, after your changes, you need to convert back to a string (e.g., to write to a file), use the string `join` method to “implode” the list back into a string:

```
>>> S = ''.join(L)
>>> S
'spaxxy'
```

The `join` method may look a bit backward at first sight. Because it is a method of strings (not of lists), it is called through the desired delimiter. `join` puts the list's strings together, with the delimiter between list items; in this case, it uses an empty string delimiter to convert from a list back to a string. More generally, any string delimiter and list of strings will do:

```
>>> 'SPAM'.join(['eggs', 'sausage', 'ham', 'toast'])
'eggsSPAMsausageSPAMhamSPAMtoast'
```

## String Method Examples: Parsing Text

Another common role for string methods is as a simple form of text *parsing*—that is, analyzing structure and extracting substrings. To extract substrings at fixed offsets, we can employ slicing techniques:

```
>>> line = 'aaa bbb ccc'
>>> col1 = line[0:3]
>>> col3 = line[8:]
>>> col1
'aaa'
>>> col3
'ccc'
```

Here, the columns of data appear at fixed offsets, and so may be sliced out of the original string. This technique passes for parsing, as long as the components of your data have fixed positions. If instead some sort of delimiter separates the data, you



can pull out its components by splitting. This will work even if the data may show up at arbitrary positions within the string:

```
➔ >>> line = 'aaa bbb ccc'
>>> cols = line.split()
>>> cols
['aaa', 'bbb', 'ccc']
```

The string `split` method chops up a string into a list of substrings, around a delimiter string. We didn't pass a delimiter in the prior example, so it defaults to whitespace—the string is split at groups of one or more spaces, tabs, and newlines, and we get back a list of the resulting substrings. In other applications, more tangible delimiters may separate the data. This example splits (and hence parses) the string at commas, a separator common in data returned by some database tools:

```
➔ >>> line = 'bob,hacker,40'
>>> line.split(',')
['bob', 'hacker', '40']
```

Delimiters can be longer than a single character, too:

```
➔ >>> line = "i'mSPAMaSPAMlumberjack"
>>> line.split("SPAM")
["i'm", 'a', 'lumberjack']
```

Although there are limits to the parsing potential of slicing and splitting, both run very fast, and can handle basic text-extraction chores.

## Other Common String Methods in Action

Other string methods have more focused roles—for example, to strip off whitespace at the end of a line of text, perform case conversions, test content, and test for a substring at the end:

```
➔ >>> line = "The knights who sy Ni!\n"
>>> line.rstrip()
'The knights who sy Ni!'
>>> line.upper()
'THE KNIGHTS WHO SY NI!\n'
>>> line.isalpha()
False
>>> line.endswith('Ni!\n')
True
```

Alternative techniques can also sometimes be used to achieve the same results as string methods—the `in` membership operator can be used to test for the presence of a substring, for instance, and length and slicing operations can be used to mimic `endswith`:

```
➔ >>> line
'The knights who sy Ni!\n'

>>> line.find('Ni') != -1    # Search via method call or expression
True
```

```

>>> 'Ni' in line
True

>>> sub = 'Ni!\n'
>>> line.endswith(sub)      # End test via method call or slice
True
>>> line[-len(sub):] == sub
True

```

Because there are so many methods available for strings, we won't look at every one here. You'll see some additional string examples later in this book, but for more details, you can also turn to the Python library manual and other documentation sources, or simply experiment interactively on your own.

Note that none of the string methods accepts patterns—for pattern-based text processing, you must use the Python `re` standard library module, an advanced tool that was introduced in Chapter 4, but is mostly outside the scope of this text. Because of this limitation, though, string methods sometimes run more quickly than the `re` module's tools.

## The Original string Module

The history of Python's string methods is somewhat convoluted. For roughly the first decade of Python's existence, it provided a standard library module called `string` that contained functions that largely mirror the current set of string object methods. In response to user requests, in Python 2.0, these functions were made available as methods of string objects. Because so many people had written so much code that relied on the original `string` module, however, it was retained for backward compatibility.

Today, you should use only string methods, not the original `string` module. In fact, the original module-call forms of today's string methods are scheduled to be deleted from Python in Release 3.0, due out soon after this edition is published. However, because you may still see the module in use in older Python code, a brief look is in order here.

The upshot of this legacy is that in Python 2.5, there technically are still two ways to invoke advanced string operations: by calling object methods, or by calling `string` module functions, and passing in the object as an argument. For instance, given a variable `X` assigned to a string object, calling an object method:

➡ `X.method(arguments)`

is usually equivalent to calling the same operation through the `string` module (provided that you have already imported the module):

➡ `string.method(X, arguments)`

Here's an example of the method scheme in action:

➡ 


```

>>> S = 'a+b+c+'
>>> x = S.replace('+', 'spam')

```

```
>>> x
'aspambspamcspam'
```

To access the same operation through the `string` module, you need to import the module (at least once in your process) and pass in the object:



```
>>> import string
>>> y = string.replace(x, '+', 'spam')
>>> y
'aspambspamcspam'
```

Because the module approach was the standard for so long, and because strings are such a central component of most programs, you will probably see both call patterns in Python code you come across.

Again, though, today you should use method calls instead of the older module calls. There are good reasons for this, besides the fact that the module calls are scheduled to go away in Release 3.0. For one thing, the module call scheme requires you to import the `string` module (methods do not require imports). For another, the module makes calls a few characters longer to type (when you load the module with `import`, that is, not using `from`). And, finally, the module runs more slowly than methods (the current module maps most calls back to the methods and so incurs an extra call along the way).

The original `string` module will probably be retained in Python 3.0 because it contains additional tools, including predefined string constants, and a template object system (an advanced tool omitted here—see the Python library manual for details on template objects). Unless you really want to change your code when 3.0 rolls out, though, you should consider the basic string operation calls in it to be just ghosts from the past.

## General Type Categories

Now that we've explored the first of Python's collection objects, the string, let's pause to define a few general type concepts that will apply to most of the types we look at from here on. With regard to built-in types, it turns out that operations work the same for all the types in the same category, so we'll only need to define most of these ideas once. We've only examined numbers and strings so far, but because they are representative of two of the three major type categories in Python, you already know more about other types than you might think.

### Types Share Operation Sets by Categories

As you've learned, strings are immutable sequences: they cannot be changed in-place (the *immutable* part), and they are positionally ordered collections that are accessed by offset (the *sequence* part). Now, it so happens that all the sequences we'll study in this part of the book respond to the same sequence operations shown in this chapter

at work on strings—concatenation, indexing, iteration, and so on. More formally, there are three type (and operation) categories in Python:

#### *Numbers*

Support addition, multiplication, etc.

#### *Sequences*

Support indexing, slicing, concatenation, etc.

#### *Mappings*

Support indexing by key, etc.

We haven't yet explored mappings on our in-depth tour (dictionaries are discussed in the next chapter), but the other types we encounter will mostly be more of the same. For example, for any sequence objects *X* and *Y*:

- *X* + *Y* makes a new sequence object with the contents of both operands.
- *X* \* *N* makes a new sequence object with *N* copies of the sequence operand *X*.

In other words, these operations work the same on any kind of sequence, including strings, lists, tuples, and some user-defined object types. The only difference is that the new result object you get back is the same type as the operands *X* and *Y*—if you concatenate lists, you get back a new list, not a string. Indexing, slicing, and other sequence operations work the same on all sequences, too; the type of the objects being processed tells Python which task to perform.

## **Mutable Types Can Be Changed In-Place**

The immutable classification is an important constraint to be aware of, yet it tends to trip up new users. If an object type is immutable, you cannot change its value in-place; Python raises an error if you try. Instead, you must run code to make a new object containing the new value. Generally, immutable types give some degree of integrity by guaranteeing that an object won't be changed by another part of a program. For a refresher on why this matters, see the discussion of shared object references in Chapter 6.

## **Chapter Summary**

In this chapter, we took an in-depth tour of the string object type. We learned about coding string literals, and explored string operations, including sequence expressions, string formatting, and string method calls. Along the way, we studied a variety of concepts in depth, such as slicing, method calls, and triple-quoted block strings. We also defined some core ideas common to a variety of types: sequences, for example, share an entire set of operations. In the next chapter, we'll continue our types tour with a look at the most general object collections in Python—the list and dictionary. As you'll find, much of what you've learned here will apply to those types as well. First, though, here's another chapter quiz to review the material introduced here.

## Chapter Quiz

1. Can the string `find` method be used to search a list?
2. Can a string slice expression be used on a list?
3. How would you convert a character to its ASCII integer code? How would you convert the other way, from an integer to a character?
4. How might you go about changing a string in Python?
5. Given a string `S` with the value `"s,pa,m"`, name two ways to extract the two characters in the middle.
6. How many characters are there in the string `"a\nb\x1f\000d"`?
7. Why might you use the `string` module instead of string method calls?

## Quiz Answers

1. No, because methods are always type-specific; that is, they only work on a single data type. Expressions are generic, though, and may work on a variety of types. In this case, for instance, the `in` membership expression has a similar effect, and can be used to search both strings and lists.
2. Yes. Unlike methods, expressions are generic, and apply to many types. In this case, the slice expression is really a sequence operation—it works on any type of sequence object, including strings, lists, and tuples. The only difference is that when you slice a list, you get back a new list.
3. The built-in `ord(S)` function converts from a one-character string to an integer character code; `chr(I)` converts from the integer code back to a string.
4. Strings cannot be changed; they are immutable. However, you can achieve a similar effect by creating a new string—by concatenating, slicing, running formatting expressions, or using a method call like `replace`—and then assigning the result back to the original variable name.
5. You can slice the string using `S[2:4]`, or split on the comma and index the string using `S.split(',')[1]`. Try these interactively to see for yourself.
6. Six. The string `"a\nb\x1f\000d"` contains the bytes `a`, newline (`\n`), `b`, binary 31 (a hex escape `\x1f`), binary 0 (an octal escape `\000`), and `d`. Pass the string to the built-in `len` function to verify this, and print each of its characters' `ord` results to see the actual byte values. See Table 7-2 for more details.
7. You should never use the `string` module instead of string object method calls today—it's deprecated, and its calls are slated for removal in Python 3.0. The only reason for using the `string` module at all is for its other tools, such as pre-defined constants, and advanced template objects.