Testing

It is often necessary in processing a string to determine what kinds of characters it "contains." These methods provide convenient ways to determine that for the most common cases:

`str1.isalpha()`
: Returns true if *str1* is not empty and all of its characters are alphabetic

`str1.isalnum()`
: Returns true if *str1* is not empty and all of its characters are alphanumeric

`str1.isdigit()`
: Returns true if *str1* is not empty and all of its characters are digits

`str1.numeric()`
: Returns true if *str1* is not empty and all of its characters are numeric, including Unicode number values and names

`str1.isdecimal()`
: Returns true if *str1* is not empty and all of its characters are characters that can be used in forming decimal-radix numbers, including Unicode number values and names

`str1.islower()`
: Returns true if *str1* contains at least one "cased" character and all of its cased characters are lowercase

`str1.isupper()`
: Returns true if *str1* contains at least one "cased" character and all of its cased characters are uppercase

`str1.istitle()`
: Returns true if *str1* is not empty, all of its uppercase characters follow "uncased" characters (spaces, punctuation, etc.), and all of its lowercase characters follow cased characters

Searching

The following methods search for one string inside another. In the descriptions that follow, *startpos* and *endpos* function as they do in slices:

`str1.startswith(str2[, startpos, [endpos]])`
: Returns true if *str1* starts with *str2*

`str1.endswith(str2[, startpos, [endos]])`
: Returns true if *str1* ends with *str2*

`str1.find(str2[, startpos[, endpos]])`
: Returns the lowest index of *str1* at which *str2* is found, or `-1` if it is not found

str1.rfind(*str2*[, *startpos*[, *endpos*]])
Performs a reverse find: returns the *highest* index where *str2* is found in *str1*, or -1 if it is not found

str1.index(*str2*[, *startpos*[, *endpos*]])
Returns the lowest index of *str1* at which *str2* is found, or ValueError if it is not found

str1.rindex(*str2*[, *startpos*[, *endpos*]])
Returns the *highest* index of str1 at which str2 is found, or ValueError if it is not found

str1.count(*str2*[, *startpos*[, *endpos*]])
Returns the number of occurrences of *str2* in *str1*

Replacing

Methods that return a new string with parts of the old string replaced with something else form the basis of a lot of code. The replace method is used particularly frequently, but there are also two other methods that constitute a powerful little facility (though they aren't used all that much):

str1.replace(*oldstr*, *newstr*[, *count*])
Returns a copy of *str1* with all occurrences of the substring *oldstr* replaced by the string *newstr*; if *count* is specified, only the first *count* occurrences are replaced.

str1.translate(*dictionary*)
With *dictionary* having integers as keys, returns a copy of *str1* with any character *char* for which ord(*char*) is a key in *dictionary* replaced by the corresponding value. Exactly what the replacement does depends on the type of the value in the dictionary, as follows:

None
Character is removed from *str1*

Integer n
Character is replaced by chr(*n*)

String str2
Character is replaced by *str2*, which may be of any length

str.maketrans(*x*[, *y*[, *z*]])
(Called directly through the str type, not an individual string.) Produces a translation table for use with translate more conveniently than manually constructing the table. Arguments are interpreted differently depending on how many there are:

x
x is a dictionary like that expected by translate, except that its keys may be either integers or one-character strings.

x, *y*

x and *y* are strings of equal length; the table will translate each character of *x* to the character in the corresponding position of *y*.

x, *y*, *z*

As with two arguments, plus all characters in the string *z* will be translated to *x* (i.e., removed).

Changing case

The methods listed in this section return a new string that is a copy of the original with its characters converted as necessary to a specified case:

`str1.lower()`

Returns a copy of the string with all of its characters converted to lowercase

`str1.upper()`

Returns a copy of the string with all of its characters converted to uppercase

`str1.capitalize()`

Returns a copy of the string with only its first character capitalized; has no effect if the first character is not a letter (e.g., if it is a space)

`str1.title()`

Returns a copy of the string with each word beginning with an uppercase character and the rest lowercase

`str1.swapcase()`

Returns a copy of the string with lowercase characters made uppercase and vice versa

Reformatting

Each of the methods in this group returns a string that is a copy of the original after applying text formatting operations:

`str1.lstrip([chars])`

Returns a copy of `str1` with leading characters removed. *chars* is a string that includes the characters to be removed (any combination of them); if it is omitted or `None`, the default is whitespace.

`str1.rstrip([chars])`

Returns a copy of `str1` with trailing characters removed. *chars* is a string that includes the characters to be removed (any combination of them); if it is omitted or `None`, the default is whitespace.

`str1.strip([chars])`

Returns a copy of `str1` with leading and trailing characters removed. *chars* is a string that includes the characters to be removed (any combination of them); if it is omitted or `None`, the default is whitespace.

Assignment expression	Result
`lst[len(lst):len(lst)] = coll`	Adds the elements of *coll* at the end of *lst*
`lst += coll`	
`lst[:] = coll`	Replaces the entire contents of *lst* with the elements of *coll*

There is a simple statement that can remove elements: del, for "delete."

STATEMENT

Deletion

The del statement removes one or more elements from a list or bytearray.

```
del lst[n]      # remove the nth element from lst
del lst[i:j]    # remove the ith through jth elements from lst
del lst[i:j:k]  # remove every k elements from i up to j from lst
```

List modification methods

Table 3-12 shows the methods that change a list. These methods are unusual in that they actually change the list itself, rather than producing a modified copy as would similar methods of other types. They are also unusual because—with the exception of pop—they do not return a value (i.e., they return None).

Table 3-12. List modification methods

Method	Result
`lst.append(x)`	Adds *x* to the end of *lst*
`lst.extend(x)`	Adds the elements of *x* at the end of *lst*
`lst.insert(i, x)`	Inserts *x* before the i^{th} element of *lst*
`lst.remove(x)`	Removes the first occurrence of *x* from *lst*; an error is raised if *x* is not in *lst*
`lst.pop([i])`	Removes the i^{th} element from *lst* and returns it; if *i* is not specified, removes the last element
`lst.reverse()`	Reverses the list
`lst.sort([reverseflag] [, keyfn])`	Sorts the list by comparing elements or, if *keyfn* is included in the call, comparing the results of calling *keyfn* for each element; if *reverseflag* is true, the ordering is reversed; *keyfn* and *reverseflg* must be specified as keyword arguments, not positionally[a]

[a] Atypically, the optional arguments to sort may not be supplied positionally. Either or both may be supplied, but only as keyword arguments. The parameter reverse is simply a flag that controls whether the list is sorted in increasing or decreasing order. The *keyfn* parameter is explained at the end of the chapter, in the section "Functional Parameters" on page 89.

Sets, frozensets, strings, tuples, and lists can be concatenated with other values of the same type, and the result is a new value of the same type. List modifications are different. No new list is created; instead, the contents of the original list are changed. This is particularly evident in the different results obtained by concatenating two lists as

opposed to using
Figure 3-1:

```
>>> list1 = [1
>>> list2 = [4
>>> list1 + li
[1, 2, 3, 4, 5
>>> list1
[1, 2, 3]
>>> list2
[4, 5]
>>> list1.exte
>>> list1
[1, 2, 3, 4, 5
>>> list2
[4, 5]
```

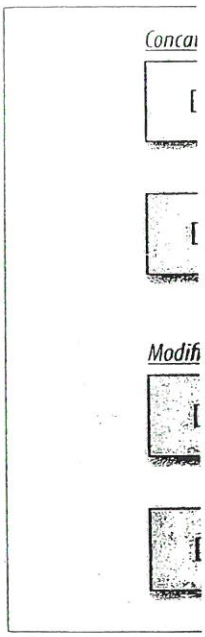
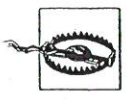


Figure 3-1. List con



Aug
lefth

Sequence-oriented

There are some i
included earlier i
they take as argu

usually lists, but in principle they can be any kind of sequence. Remember too that when a sequence argument is expected, a string is interpreted as a sequence of one-character strings. Here are the remaining string methods:

string`.splitlines(`[*keepflg*]`)`
: Returns a list of the "lines" in *string*, splitting at end-of-line characters. If *keepflg* is omitted or is false, the end-of-line characters are not included in the lines; otherwise, they are.

string`.split(`[*sepr*[`,` *maxwords*]]`)`
: Returns a list of the "words" in *string*, using *sepr* as a word delineator. In the special case where *sepr* is omitted or is `None`, words are delineated by any consecutive whitespace characters; if *maxwords* is specified the result will have at most *maxwords* +1 elements.

string`.rsplit(`[*sepr*[`,` *maxwords*]]`)`
: Performs a reverse split: same as `split` except that if *maxwords* is specified and its value is less than the number of words in *string* the result returned is a list containing the *last* *maxwords*+1 words.

sepr`.join(`*seq*`)`
: Returns a string formed by concatenating the strings in *seq* separated by *sepr*, which can be any string (including the empty string).

string`.partition(`*sepr*`)`
: Returns a tuple with three elements: the portion of *string* up to the first occurrence of *sepr*, *sepr*, and the portion of *string* after the first occurrence of *sepr*. If *sepr* is not found in *string*, the tuple is (*string*, `''`, `''`).

string`.rpartition(`*sepr*`)`
: Returns a tuple with three elements: the portion of *string* up to the last occurrence of *sepr*, *sepr*, and the portion of *string* after the last occurrence of *sepr*. If *sepr* is not found in *string*, the tuple is (`''`, `''`, *string*).

Mappings

A *mapping* is a *mutable unordered* collection of *key/value pairs*.† Computer scientists use a number of other names to refer to data structures implementing mappings, including *associative arrays*, *lookup tables*, and *hash tables*. A physical dictionary is a real-world example of a mapping: given a word, you get a definition. Figure 3-2 illustrates the concept.

† The term "mapping" comes from mathematics, where it represents a function from a "domain" of values to a "range" of values.

Figure 3-2. A dicti

Dictionaries

Dictionaries hav
have sets, frozer
your programs u
in Python: dict,
collection argun
those elements r

```
dict((('A', '
      ('C', '
      ('G', '
      ('T', '
     ))
```

Because they are
to that of sets:‡ a
each key and va
before and after
colon, so you mi
(not an empty s
implementation

```
>>> {'A': 'ad
{'A': 'adenin
```

The keys of a m
has no way to di
implementation

‡ You could consider
they are often imple

Slice assignment can remove an element from a mutable sequence, so it isn't necessary to use `del` statements with mutable sequences (though they can make your code clearer). Dictionary assignment, however, can only add or replace an element, not remove one. That makes `del` statements more important with dictionaries than with sequences. While there are `dict` methods that remove elements from a dictionary, `del` statements are more concise.

Dictionary methods

Because items in a mapping involve both a key and an associated value, there are methods that deal with keys, values, and key/value pairs. In addition, there are methods that perform the equivalent of the dictionary operations but with more options. There is also a method for adding the key/value pairs from another dictionary, replacing the values of any keys that were already present. Explanations of the details of these operations are found in Table 3-15.

Table 3-15. Dictionary methods

Expression	Result
`d.get(key[, default_value])`	Like `d[key]`, but does not cause an error if `d` does not contain `key`; instead, it returns `default_value`, which, if not provided, is None
`d.setdefault(key[, default_value])`	Like `d[key]` if `key` is in `d`; otherwise, adds `key` with a value of `default_value` to the dictionary and returns `default_value` (if not specified, `default_value` is None)
`d.pop(key[, default_value])`	Like `del d[key]`, but does not cause an error if `d` does not contain `key`; instead, it returns `default_value`, which, if not provided, is None
`d1.update(d2)`	For each key in `d2`, sets `d1[key]` to `d2[key]`, replacing the existing value if there was one
`d.keys()`	Returns a special sequence-like object containing the dictionary's keys
`d.values()`	Returns a special sequence-like object containing the dictionary's values
`d.items()`	Returns a special sequence-like object containing (`key`, `value`) tuples for the dictionary's keys

Note that the last three methods return "sequence-like objects": they aren't sequences, but they can be used as if they were in many contexts. If you need a dictionary's keys, values, or items in the form of a list, simply call `list` with what the corresponding method returns.

Streams

A *stream* is a *temporally ordered* sequence of *indefinite length*, usually limited to one type of element. Each stream has two ends: a *source* that provides the elements and a *sink* that absorbs the elements. The term "stream" is apt, conjuring as it does the flow

of water into or o
network connectio
network sources a

Your input to a c
characters (see Fig
characters. These
don't necessary e
stream itself. For
events that happe
buffer that does h
of efficiency and
type.)

The

Figure 3-3. A stream

Files

Usually, the term
The term "file" is
A Python file is a
objects provide m
terface is sufficien
data they represe

The smallest unit
a 0. *Bytes* are gro
through the vario
do get grouped in
systems, but prog

STATEMENT

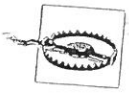
The with Statement

The with statement is used to open and name a file, then automatically close the file regardless of whether an error occurs during the execution of its statements. Like a def statement, a with statement contains an indented block of statements.

```
with open(path, mode) as name:
    statements-using-name
```

More than one file can be opened with the same with statement, as when reading from one and writing to the other.

```
with open(path1, mode1) as name1, open(path2, mode2) as name2, ...:
    statements-using-names
```



In versions of Python before 2.6, the executable first line of a file that uses a with statement must be:

```
from __future__ import with_statement
```

File methods

Methods for reading from files include the following:

fileobj.read([*count*])
: Reads *count* bytes, or until the end of the file, whichever comes first; if *count* is omitted, reads everything until the end of the file. If at the end of the file, returns an empty string. This method treats the file as an input stream of characters.

fileobj.readline([*count*])
: Reads one line from the file object and returns the entire line, including the end-of-line character; if *count* is present, reads at most *count* characters. If at the end of the file, returns an empty string. This method treats the file as an input stream of lines.

fileobj.readlines()
: Reads lines of a file object until the end of the file is reached and returns them as a list of strings; this method treats the file as an input stream of lines.

File methods for writing to files include the following:

fileobj.write(*string*)
: Writes *string* to *fileobj*, treating it as an output stream of characters.

fileobj.writelines(*sequence*)
: Writes each element of *sequence*, which must all be strings, to *fileobj*, treating it as an output stream of lines. Note, however, that although this method's name is intentionally analogous to readlines, newline characters are not added to the strings in *sequence* when they are written to *fileobj*.

Notice that the
characters or l
next item in th

In addition, th
documented ir
the file argum
a way to conve

Example

To manipulate
into a string ar
with large files,
at reading FAS

The first step
string at every
example. I'll er
pieces that you

```
>>> '>id1>i
['', 'id1',
```

The list returne
difficult it wou
writing a 20-lin
with small fun
lot right—with

We can captur
file for the stri
eliminates that

Example 3-6. Rec

```
def read_FASTA_s
    with open(fi
        return f
```

There are some
the description
string contains
ways to read da

#FASTA-formattec
sequences broken
with a ">" charac
identifiers and co