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.isnumeric()
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, [endpos])
Returns true if str1 ends with str2

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

```python
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
str1.maketrans(x[, y[, z]])
    (Called directly through the str type, not an individual string.) Produces a transla-
tion 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.
```

---

Changing case
The methods
its characters
str1.lower() Returns:
    str1.upper() Returns:
    str1.capitalize Returns:
        if the first
    str1.title() Returns:
        and the
    str1.swapcase Returns versa

Reformatting
Each of the:
applying tex
str1.lstrip
    Returns
    cludes t
    None, th
str1.rstrip
    Returns
    cludes t
    None, th
str1.strip
    Returns
    string d
    is omitted
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.title()` Returns a copy of the string with each word beginning with an uppercase character and the rest lowercase.
- `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., `It is a space`).

As with two arguments, plus all characters in the string will be translated to the character in the corresponding position of

\[ x, y \]

where `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`, translated.

x, y
Assignment expression

\[
\text{lst}[\text{len(lst)}: \text{len(lst)}] = \text{call}
\]

Result

Adds the elements of \text{call} at the end of \text{lst}

\[
\text{lst} += \text{call}
\]

Replaces the entire contents of \text{lst} with the elements of \text{call}

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

**STATEMENT**

**Deletion**

The \text{del} statement removes one or more elements from a list or bytestring.

\[
\begin{align*}
\text{del lst} & & \# \text{ remove the } n \text{th element from lst} \\
\text{del lst}[i:j] & & \# \text{ remove the } i \text{th through } j \text{th elements from lst} \\
\text{del lst}[i:j:k] & & \# \text{ remove every } k \text{ elements from } i \text{ up to } j \text{ from lst}
\end{align*}
\]

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 \text{pop}—they do not return a value (i.e., they return \text{None}).

**Table 3-12. List modification methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{lst.append(x)}</td>
<td>Adds \text{x} to the end of \text{lst}</td>
</tr>
<tr>
<td>\text{lst.extend(x)}</td>
<td>Adds the elements of \text{x} at the end of \text{lst}</td>
</tr>
<tr>
<td>\text{lst.insert(i, x)}</td>
<td>Inserts \text{x} before the \text{i}th element of \text{lst}</td>
</tr>
<tr>
<td>\text{lst.remove(x)}</td>
<td>Removes the first occurrence of \text{x} from \text{lst}; an error is raised if \text{x} is not in \text{lst}</td>
</tr>
<tr>
<td>\text{lst.pop(i)}</td>
<td>Removes the \text{i}th element from \text{lst} and returns it; if \text{i} is not specified, removes the last element</td>
</tr>
<tr>
<td>\text{lst.reverse()}</td>
<td>Reverses the list</td>
</tr>
<tr>
<td>\text{lst.sort(key=\text{keyfn}[i, keyfn]])}</td>
<td>Sorts the list by comparing elements or, if \text{keyfn} is included in the call, comparing the results of calling \text{keyfn} for each element; if \text{reverseflag} is true, the ordering is reversed; \text{keyfn} and \text{reverseflag} must be specified as keyword arguments, not positionally\footnote{Atypically, the optional arguments to \text{sort} may be supplied positionally. Either or both may be supplied, but only as keyword arguments. The parameter \text{reverse} is simply a flag that controls whether the list is sorted in increasing or decreasing order. The \text{keyfn} parameter is explained at the end of the chapter, in the section &quot;Functional Parameters&quot; on page 89.}</td>
</tr>
</tbody>
</table>

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:**

\[
\begin{align*}
\text{list1} & = [1] \\
\text{list2} & = [4] \\
\text{list1} & + \text{list2} = [1, 2, 3, 4, 5] \\
\text{list1} & .exte \\
\text{list1} & + \text{list2} = [1, 2, 3, 4, 5] \\
\text{list2} & \text{.exte}
\end{align*}
\]

There are some ii included earlier i they take as args
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([keepfil])
```

Returns a list of the "lines" in string, splitting at end-of-line characters. If keepfil 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.

†† You could consider they are often imp
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**

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

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 out network connections.

Your input to a `c` characters (see Fig characters. These don’t necessary e: stream itself. For events that happen `buffer` that happens of efficiency and c type.)

**Files**

Usually, the term “file” is A Python file is an objects provide m terface is sufficient data they represe The smallest unit a 0. Bytes are gro through the various do get grouped int systems, but prog
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.

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

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

```python
with open(path1, mode1) as name1, open(path2, mode2) as name2, ...
    statements
```

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

```python
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`.

Example

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

```
    >>> 'sid' + '111''
    'sid111'
```

The list returns difficulty it would writing a 20-lin with small fun lot right—with

We can capture file for the string eliminates that

Example 3-6. Rec

def read_FAS
    with open(f
```

There are some the description string contains ways to read di: