```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the table as keyword arguments with the '**' notation.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.f
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables.

For a complete overview of string formatting with `str.format()`, see *Format String Syntax*.

7.1.1. Old string formatting

The `%` operator can also be used for string formatting. It interprets the left argument much like a `sprintf()`-style format string to be applied to the right argument, and returns the string resulting from this formatting operation. For example:

```
>>> import math
>>> print('The value of PI is approximately %5.3f.' % math.pi)
The value of PI is approximately 3.142.
```

More information can be found in the *printf-style String Formatting* section.

7.2. Reading and Writing Files

`open()` returns a *file object*, and is most commonly used with two arguments: `open(filename, mode)`.

```
>>> f = open('workfile', 'w')
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The *mode* argument is optional; `'r'` will be assumed if it's omitted.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific encoding (the default being UTF-8). `'b'` appended to the mode opens the file in *binary mode*: now the data is read and written in the form of bytes objects. This mode should be used for all files that don't contain text.

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in `JPEG` or `EXE` files. Be very careful to use binary mode when reading and writing such files.

7.2.1. Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string or bytes object. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most *size* bytes are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`''`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.

`f.write(string)` writes the contents of *string* to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

To write something other than a string, it needs to be converted to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
18
```

`f.tell()` returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in *binary mode* and an opaque number when in *text mode*.

To change the file object's position, use `f.seek(offset, from_what)`. The position is computed from adding *offset* to a reference point; the reference point is selected by the *from_what* argument. A *from_what* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *from_what* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)     # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

In text files (those opened without a `b` in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with `seek(0, 2)`) and the only valid *offset* values are those returned from the `f.tell()`, or zero. Any other *offset* value produces undefined behaviour.

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file. After calling `f.close()`, attempts to use the file object will automatically fail.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent `try`-`finally` blocks:

```
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

7.2.2. Saving structured data with `json`

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation). The standard module called `json` can take Python data hierarchies, and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

Note: The JSON format is commonly used by modern applications to allow for data exchange. Many programmers are already familiar with it, which makes it a good choice for interoperability.

If you have an object `x`, you can view its JSON string representation with a simple line of code:

```
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a *text file*. So if `f` is a *text file* object opened for writing, we can do this:

```
json.dump(x, f)
```

To decode the object again, if `f` is a *text file* object which has been opened for reading:

```
x = json.load(f)
```

This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the `json` module contains an explanation of this.

See also: `pickle` - the pickle module

Contrary to JSON, *pickle* is a protocol which allows the serialization of arbitrarily complex Python objects. As such, it is specific to Python and cannot be used to communicate with applications written in other languages. It is also insecure by default: deserializing pickle data coming from an untrusted source can execute arbitrary code, if the data was crafted by a skilled attacker.