11 Files

Key terms: `with open read write close bytes`

Reading: Severance 7

Exercise: Write a program that creates a file with a randomly chosen filename.

Operating systems store files in file systems using addresses called pathnames, or paths. Absolute pathnames begin at the filesystem root, so in UNIX, they start with a slash: `/proc/version`. Relative pathnames begin at some arbitrary place in the hierarchy, making them inherently ambiguous: `letter.txt`.

Files are inherently sequential storage, which matches Python's model of iterables. An open file is represented by a "wrapper" object, through which line-oriented files can be handled directly: the wrapper is an iterable object, returning one line at a time. Rarely is it wise to read a file more than once; in the event we want continual access, we might prefer to keep parts of it in memory.

A call to `open()` yields a wrapper object for dealing with the contents of a file on disk. It is convenient to use the `with` keyword for a "context manager" that closes the file for us. The wrapper object has `read()`, `write()`, and `seek()`, methods. Permissions rules administered by the operating system control access to files; the methods for reading and writing permissions modes are in the `os` module. File accesses is for reading only by default, or else writing, appending, and so on based on a mode parameter.

```
# Store the contents of a file as a single string:
text = open("/proc/version").read()

# Store the contents of a file in a list of strings:
lines = open("/etc/passwd").readlines()
```

Since several sorts of things can go wrong with file access, it is wise to run robust exception tests to check for various errors:

```
# Try to open the file specified:

import sys
try:
 file = open(sys.argv[1])
except IndexError:
 print ('No filename passed.')
except FileNotFoundError:
 print ('File not found.')
except PermissionError:
 print ('File not readable.')
```

Python's default encoding is UTF-8: strings are stored with Unicode codepoints, represented in output using the UTF-8 scheme. Python 3 is righteously strict about encodings, which prevents confusion between strings and encoded streams (`bytes`). This change enables Python code to handle all scripts with no fuss. The above examples will raise a `UnicodeDecodeError` if the input file is not valid UTF-8. This behavior is the alternative to the ostrich-style decoding you may have seen in web pages and emails seemingly containing gibberish.

```
# Try to read UTF-8 text from an open file:

try:
 text = file.read()
except UnicodeDecodeError:
 print ('File is not in UTF-8.')
```

Next, we will address functions.