Reading From Text Files

WE'LL COVER THE FOLLOWING

Before you can read from a file, you need to open it. Opening a file in Python couldn't be easier:

```
a_file = open('chinese.txt', encoding='utf-8')
```

Python has a built-in `open()` function, which takes a filename as an argument. Here the filename is `'chinese.txt'`. There are five interesting things about this filename:

- It's not just the name of a file; it's a combination of a directory path and a filename. A hypothetical file-opening function could have taken two arguments — a directory path and a filename — but the `open()` function only takes one. In Python, whenever you need a "filename," you can include some or all of a directory path as well.
- The directory path uses a forward slash, but I didn't say what operating system I was using. Windows uses backward slashes to denote subdirectories, while Mac OS X and Linux use forward slashes. But in Python, forward slashes always Just Work, even on Windows.
- The directory path does not begin with a slash or a drive letter, so it is

called a *relative path*. Relative to what, you might ask? Patience, grasshopper.

- It's a string. All modern operating systems (even Windows!) use Unicode to store the names of files and directories. Python 3 fully supports non-ASCII pathnames.
- It doesn't need to be on your local disk. You might have a network drive mounted. That "file" might be a figment of an entirely virtual filesystem. If your computer considers it a file and can access it as a file, Python can open it.

But that call to the `open()` function didn't stop at the filename. There's another argument, called `encoding`. Oh dear, that sounds dreadfully familiar.

Character Encoding Rears Its Ugly Head

Bytes are bytes; characters are an abstraction. A string is a sequence of Unicode characters. But a file on disk is not a sequence of Unicode characters; a file on disk is a sequence of bytes. So if you read a "text file" from disk, how does Python convert that sequence of bytes into a sequence of characters? It decodes the bytes according to a specific character encoding algorithm and returns a sequence of Unicode characters (otherwise known as a string).

```
# This example was created on Windows. Other platforms may
# behave differently, for reasons outlined below.
file = open('chinese.txt')
a_string = file.read()
#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 4, in <module>
# a_string = file.read()
# File "/usr/lib/python3.4/encodings/ascii.py", line 26, in decode
# return codecs.ascii_decode(input, self.errors)[0]
#UnicodeDecodeError: 'ascii' codec can't decode byte 0xe6 in position 17: ordinal not in rang
```

What just happened? You didn't specify a character encoding, so Python is forced to use the default encoding. What's the default encoding? If you look closely at the traceback, you can see that it's dying in `cp1252.py`, meaning that Python is using CP-1252 as the default encoding here. (CP-1252 is a common encoding on computers running Microsoft Windows.) The CP-1252 character set doesn't support the characters that are in this file, so the read fails with an

ugly `UnicodeDecodeError`.

The default encoding is platform-dependent.

But wait, it's worse than that! The default encoding is *platform-dependent*, so this code *might* work on your computer (if your default encoding is utf-8), but then it will fail when you distribute it to someone else (whose default encoding is different, like CP-1252).

> *If you need to get the default character encoding, import the **locale** module and call **locale.getpreferredencoding()**. On my Windows laptop, it returns **'cp1252'**, but on my Linux box upstairs, it returns **'UTF8'**. I can't even maintain consistency in my own house! Your results may be different (even on Windows) depending on which version of your operating system you have installed and how your regional/language settings are configured. This is why it's so important to specify the encoding every time you open a file.*

Stream Objects

So far, all we know is that Python has a built-in function called `open()`. The `open()` function returns a *stream object*, which has methods and attributes for getting information about and manipulating a stream of characters.

```
a_file = open('chinese.txt', encoding='utf-8')
print (a_file.name)                                    #①
#/usercode/chinese.txt

print (a_file.encoding)                                #②
#utf-8

print (a_file.mode )                                   #③
#r
```

① The name attribute reflects the name you passed in to the `open()` function when you opened the file. It is not normalized to an absolute pathname.

② Likewise, `encoding` attribute reflects the encoding you passed in to the

② Likewise, `encoding` attribute reflects the encoding you passed in to the `open()` function. If you didn't specify the encoding when you opened the file (bad developer!) then the `encoding` attribute will reflect `locale.getpreferredencoding()`.

③ The `mode` attribute tells you in which mode the file was opened. You can pass an optional `mode` parameter to the `open()` function. You didn't specify a mode when you opened this file, so Python defaults to '`r`', which means "open for reading only, in text mode." As you'll see later in this chapter, the file mode serves several purposes; different modes let you write to a file, append to a file, or open a file in binary mode (in which you deal with bytes instead of strings).

> *The documentation for the **open()** function lists all the possible file modes.*

Reading Data From A Text File

After you open a file for reading, you'll probably want to read from it at some point.

```
a_file = open('chinese.txt', encoding='utf-8')
a_file.read()                                                    #①
#'Dive Into Python 是为有经验的程序员编写的一本 Python 书。\n'

print (a_file.read() )                                           #②
#''
```

① Once you open a file (with the correct encoding), reading from it is just a matter of calling the stream object's `read()` method. The result is a string.

② Perhaps somewhat surprisingly, reading the file again does not raise an exception. Python does not consider reading past end-of-file to be an error; it simply returns an empty string.

Always specify an **encoding** parameter when you open a file.

What if you want to re-read a file?

```
# continued from the previous example
a_file = open('chinese.txt', encoding='utf-8')
a_file.read()                                       #①
#''

print (a_file.seek(0))                              #②
#0

print (a_file.read(16))                             #③
#'Dive Into Python'

print (a_file.read(1))                              #④
#' '

a_file.read(1)
#'是'

print (a_file.tell())                               #⑤
#20
```

① Since you're still at the end of the file, further calls to the stream object's `read()` method simply return an empty string.

② The `seek()` method moves to a specific byte position in a file.

③ The `read()` method can take an optional parameter, the number of characters to read.

④ If you like, you can even read one character at a time.

⑤ 16 + 1 + 1 = ... 20?

Let's try that again.

```
# continued from the previous example
a_file = open('chinese.txt', encoding='utf-8')
print (a_file.seek(17))                             #①
#17

a_file.read(1)                                      #②
#'是'

print (a_file.tell())                               #③
#20
```

① Move to the 17th byte.

② Read one character.

③ Now you're on the 20th byte.

Do you see it yet? The `seek()` and `tell()` methods always count bytes, but since you opened this file as text, the `read` method counts *characters*. Chinese characters require multiple bytes to encode in UTF-8. The English characters in the file only require one byte each, so you might be misled into thinking that the `seek()` and `read()` methods are counting the same thing. But that's only true for some characters.

But wait, it gets worse!

```
a_file = open('chinese.txt', encoding='utf-8')
print (a_file.seek(18))                          #①
#18

print (a_file.read(1) )                          #②
#Traceback (most recent call last):
# File "__ed_file.py", line 5, in <module>
# print (a_file.read(1) ) #\u2461
# File "/usr/lib/python3.4/codecs.py", line 313, in decode
# (result, consumed) = self._buffer_decode(data, self.errors, final)
#UnicodeDecodeError: 'utf-8' codec can't decode byte 0x98 in position 0: invalid start byte
```

① Move to the 18th byte and try to read one character.

② Why does this fail? Because there isn't a character at the 18th byte. The nearest character starts at the 17th byte (and goes for three bytes). Trying to read a character from the middle will fail with a `UnicodeDecodeError`.

Closing Files

Open files consume system resources, and depending on the file mode, other programs may not be able to access them. It's important to close files as soon as you're finished with them.

```
# continued from the previous example
a_file = open('chinese.txt', encoding='utf-8')
a_file.close()
```

Well that was *anticlimactic*.

The stream object `a_file` still exists; calling its `close()` method doesn't destroy the object itself. But it's not terribly useful.

```
# continued from the previous example
a_file = open('chinese.txt', encoding='utf-8')
print (a_file.read())                           #①
#Traceback (most recent call last):
# File "__ed_file.py", line 3, in <module>
# print (a_file.read()) # \u2460
#UnicodeEncodeError: 'ascii' codec can't encode characters in position 17-30: ordinal not in
```

```
# continued from the previous example
a_file = open('chinese.txt', encoding='utf-8')

print (a_file.seek(0))                          #②
#0
```

```
# continued from the previous example
a_file = open('chinese.txt', encoding='utf-8')

print (a_file.tell())                           #③
#0
```

```
# continued from the previous example
a_file = open('chinese.txt', encoding='utf-8')

print (a_file.close())                          #④
print (a_file.closed)                           #⑤
#True
```

① You can't read from a closed file; that raises an `IOError` exception.

② You can't seek in a closed file either.

③ There's no current position in a closed file, so the `tell()` method also fails.

④ Perhaps surprisingly, calling the `close()` method on a stream object whose file has been closed does *not* raise an exception. It's just a no-op.

⑤ Closed stream objects do have one useful attribute: the `closed` attribute will confirm that the file is closed.

Closing Files Automatically

try..finally is good. **with** is better.

Stream objects have an explicit `close()` method, but what happens if your code has a bug and crashes before you call `close()`? That file could theoretically stay open for much longer than necessary. While you're debugging on your local computer, that's not a big deal. On a production server, maybe it is.

Python 2 had a solution for this: the `try..finally` block. That still works in Python 3, and you may see it in other people's code or in older code that was ported to Python 3. But Python 2.6 introduced a cleaner solution, which is now the preferred solution in Python 3: the `with` statement.

```
with open('chinese.txt', encoding='utf-8') as a_file:
    a_file.seek(17)
    a_character = a_file.read(1)
    a_character
```

This code calls `open()`, but it never calls `a_file.close()`. The `with` statement starts a code block, like an `if` statement or a `for` loop. Inside this code block, you can use the variable `a_file` as the stream object returned from the call to `open()`. All the regular stream object methods are available — `seek()`, `read()`, whatever you need. When the with block ends, *Python calls `a_file.close()` automatically*.

Here's the kicker: no matter how or when you exit the `with` block, Python will close that file… even if you "exit" it via an unhandled exception. That's right, even if your code raises an exception and your entire program comes to a screeching halt, that file will get closed. Guaranteed.

> *In technical terms, the **with** statement creates a runtime context. In these examples, the stream object acts as a context manager. Python creates the stream object **a_file** and tells it that it is entering a runtime context. When the **with** code block is completed, Python tells the stream object that it is exiting the runtime context, and the stream object calls its own **close()** method. See Appendix B, "Classes That Can Be Used in a with Block" for details.*

There's nothing file-specific about the `with` statement; it's just a generic framework for creating runtime contexts and telling objects that they're entering and exiting a runtime context. If the object in question is a stream object, then it does useful file-like things (like closing the file automatically). But that behavior is defined in the stream object, not in the `with` statement. There are lots of other ways to use context managers that have nothing to do with files. You can even create your own, as you'll see later in this chapter.

Reading Data One Line At A Time

A "line" of a text file is just what you think it is — you type a few words and press `ENTER`, and now you're on a new line. A line of text is a sequence of characters delimited by… what exactly? Well, it's complicated, because text files can use several different characters to mark the end of a line. Every operating system has its own convention. Some use a carriage return character, others use a line feed character, and some use both characters at the end of every line.

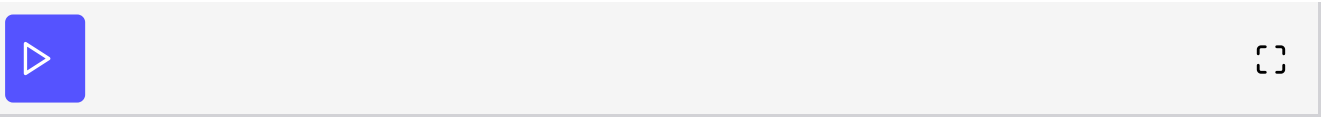
Now breathe a sigh of relief, because *Python handles line endings automatically* by default. If you say, "I want to read this text file one line at a time," Python will figure out which kind of line ending the text file uses and and it will all Just Work.

> *If you need fine-grained control over what's considered a line ending, you*

> *can pass the optional **newline** parameter to the **open()** function. See the **open()** function documentation for all the gory details.*

So, how do you actually do it? Read a file one line at a time, that is. It's so simple, it's beautiful.

```
line_number = 0
with open('favorite-people.txt', encoding='utf-8') as a_file:            #①
    for a_line in a_file:                                                #②
        line_number += 1
        print('{:>4} {}'.format(line_number, a_line.rstrip()))           #③
```



① Using the with pattern, you safely open the file and let Python close it for you.

② To read a file one line at a time, use a for loop. That's it. Besides having explicit methods like `read()`, the *stream object is also an iterator* which spits out a single line every time you ask for a value.

③ Using the `format()` string method, you can print out the line number and the line itself. The format specifier `{:>4}` means "print this argument right-justified within 4 spaces." The `a_line` variable contains the complete line, carriage returns and all. The `rstrip()` string method removes the trailing whitespace, including the carriage return characters.

```
you@localhost:~/diveintopython3$ python3 examples/oneline.py
   1 Dora
   2 Ethan
   3 Wesley
   4 John
   5 Anne
   6 Mike
   7 Chris
   8 Sarah
   9 Alex
  10 Lizzie
```

> Did you get this error?

```
you@localhost:~/diveintopython3$ python3 examples/oneline.py
Traceback (most recent call last):
  File "examples/oneline.py", line 4, in <module>
    print('{:>4} {}'.format(line_number, a_line.rstrip()))
ValueError: zero length field name in format
```

If so, you're probably using Python 3.0. You should really upgrade to Python 3.1.

> Python 3.0 supported string formatting, but only with explicitly numbered format specifiers. Python 3.1 allows you to omit the argument indexes in your format specifiers. Here is the Python 3.0-compatible version for comparison:

```
print('{0:>4} {1}'.format(line_number, a_line.rstrip()))
```