Stream Objects From Non-File Sources

WE'LL COVER THE FOLLOWING

- Handling Compressed Files

Imagine you're writing a library, and one of your library functions is going to read some data from a file. The function could simply take a filename as a string, go open the file for reading, read it, and close it before exiting. But you shouldn't do that. Instead, your api should take an *arbitrary stream object*.

In the simplest case, a stream object is anything with a `read()` method which takes an optional `size` parameter and returns a string. When called with no `size` parameter, the `read()` method should read everything there is to read from the input source and return all the data as a single value. When called with a `size` parameter, it reads that much from the input source and returns that much data. When called again, it picks up where it left off and returns the next chunk of data.

To read from a fake file, just call **read()**.

That sounds exactly like the stream object you get from opening a real file. The difference is that *you're not limiting yourself to real files*. The input source that's being "read" could be anything: a web page, a string in memory, even the output of another program. As long as your functions take a stream object and simply call the object's `read()` method, you can handle any input source that acts like a file, without specific code to handle each kind of input.

```
a_string = 'PapayaWhip is the new black.'
import io                                           #①
a_file = io.StringIO(a_string)                      #②
print (a_file.read())                               #③
#PapayaWhip is the new black.
```

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

print (a_file.seek(0))                                 #⑤
#0

print (a_file.read(10))                                #⑥
#PapayaWhip

print (a_file.tell())
#10

print (a_file.seek(18))
#18

print (a_file.read())
#new black.
```

① The `io` module defines the `StringIO` class that you can use to treat a string in memory as a file.

② To create a stream object out of a string, create an instance of the `io.StringIO()` class and pass it the string you want to use as your “file” data. Now you have a stream object, and you can do all sorts of stream-like things with it.

③ Calling the `read()` method “reads” the entire “file,” which in the case of a `StringIO` object simply returns the original string.

④ Just like a real file, calling the `read()` method again returns an empty string.

⑤ You can explicitly seek to the beginning of the string, just like seeking through a real file, by using the `seek()` method of the `StringIO` object.

⑥ You can also read the string in chunks, by passing a `size` parameter to the `read()` method.

> **io.StringIO** lets you treat a string as a text file. There’s also a **io.BytesIO** class, which lets you treat a byte array as a binary file.

Handling Compressed Files

The Python standard library contains modules that support reading and writing compressed files. There are a number of different compression schemes; the two most popular on non-Windows systems are gzip and bzip2. (You may have also encountered PKZIP archives and GNU Tar archives. Python has modules for those, too.)

The `gzip` module lets you create a stream object for reading or writing a gzip-compressed file. The stream object it gives you supports the `read()` method (if you opened it for reading) or the `write()` method (if you opened it for writing). That means you can use the methods you've already learned for regular files to *directly read or write a gzip-compressed file*, without creating a temporary file to store the decompressed data.

As an added bonus, it supports the `with` statement too, so you can let Python automatically close your gzip-compressed file when you're done with it.

```
#you@localhost:~$ python3

import gzip
with gzip.open('out.log.gz', mode='wb') as z_file:                                        #①
   z_file.write('A nine mile walk is no joke, especially in the rain.'.encode('utf-8'))

exit()
```

```
you@localhost:~$ ls -l out.log.gz                                                           ②
-rw-r--r--  1 mark mark    79 2009-07-19 14:29 out.log.gz
you@localhost:~$ gunzip out.log.gz                                                          ③
you@localhost:~$ cat out.log                                                                ④
A nine mile walk is no joke, especially in the rain.
```

① You should always open gzipped files in binary mode. (Note the `'b'` character in the `mode` argument.)

② I constructed this example on Linux. If you're not familiar with the command line, this command is showing the "long listing" of the gzip-compressed file you just created in the Python Shell. This listing shows that the file exists (good), and that it is 79 bytes long. That's actually larger than the string you started with! The gzip file format includes a fixed-length header that contains some metadata about the file, so it's inefficient for extremely small files.

③ The `gunzip` command (pronounced “gee-unzip”) decompresses the file and stores the contents in a new file named the same as the compressed file but without the `.gz` file extension.

④ The `cat` command displays the contents of a file. This file contains the string you originally wrote directly to the compressed file `out.log.gz` from within the Python Shell.

> Did you get this error?

```
with gzip.open('out.log.gz', mode='wb') as z_file:
  z_file.write('A nine mile walk is no joke, especially in the rain.'.enco
de('utf-8'))
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
AttributeError: 'GzipFile' object has no attribute '__exit__'
```

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

> Python 3.0 had a gzip module, but it did not support using a gzipped-file object as a context manager. Python 3.1 added the ability to use gzipped-file objects in a with statement.