Chapter 10

I/O Files

So far we have worked reading and writing text files. However, operating systems represent most of its files as sequences of bytes, not as text. Since reading bytes and converting them to text is a very common operation in files, Python handles the bytes by transforming the string representation with the respective *encoders/decoders*. For example, the `open` function receive the name of the file to open, but also accept as an argument the character set for encoding the bytes, and the strategy to follow when bytes are inconsistent with the format. For instance, take a look at the different methods applied to the file *example_file*:

> *# Lorem Ipsum*
>
> *Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat.*

As we mentioned above, we open a file using the `open()` function:

```
1  # 19.py
2
3  file = open('example_file', 'r', encoding='ascii', errors='replace')
```

```
print(file.read())
file.close()

# Lorem Ipsum

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut
purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.
Curabitur dictum gravida mauris. Nam arcu libero, nonummy
eget, consectetuer id, vulputate a, magna. Donec vehicula augue
eu neque. Pellentesque habitant morbi tristique senectus et netus
et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra
metus rhoncus sem. Nulla et lectus vestibulum urna fringilla
ultrices. Phasellus eu tellus sit amet tortor gravida placerat.
```

We can override and overwrite this file, using the `'w'` argument in the `open()` method as follows:

```
# 20.py

content = "Sorry but now, this file will have me."
file = open('example_file', 'w', encoding='ascii', errors='replace')
file.write(content)
file.close()
```

Now, if we re-open and read the file like at the beginning:

```
# 21.py

file = open('example_file', 'r', encoding='ascii', errors='replace')
print(file.read())
file.close()

Sorry but now, this file will have me.
```

The contents in the file changed by the last sentence we wrote using the `'w'` argument. We could instead add content at the end of the file if we replace the `'w'` by an `'a'`:

```
# 22.py

content = "\nI will be added to the end."
```

```
file = open('example_file', 'a', encoding='ascii', errors='replace')
file.write(content)
file.close()

file = open('example_file', 'r', encoding='ascii', errors='replace')
print(file.read())
file.close()
```

```
Sorry but now, this file will have me.
I will be added to the end.
```

To open a file as binary, we only need to append the char `b` to the opening mode. For example, `'wb'` and `'rb'` instead of `'w'` and `'r'`, respectively. In this case, Python opens the file like text files, but without the automatic coding from byte to text:

```
# 23.py

content = b"abcde12"
file = open('example_file_2', 'wb')
file.write(content)
file.close()

file = open('example_file_2', 'rb')
print(file.read())
file.close()
```

```
b'abcde12'
```

We can concatenate bytes by simply using the `sum` operator. In the example below, we build dynamic content in each iteration. Then it is written in an explicit bytes file.

```
# 24.py

num_lines = 100

file = open('example_file_3', 'wb')
for i in range(num_lines):
    # To the bytes function we should pass an iterable with the content to
```

```
    # convert. For this reason we pass the integer inside the list
    content = b"line_" + bytes([i]) + b" abcde12"
    file.write(content)
file.close()
```

To see the result, we re-read a fixed amount of bytes from the same file:

```
file = open('example_file_3', 'rb')
# The number 40 indicates the number of bytes that will be read from the file
print(file.read(40))
file.close()
```

```
b'line_\x00 abcde12line_\x01 abcde12line_\x02 abcde'
```

10.1 Context Manager

Every time we open a file, binary or not, we have to ensure our program close it correctly after reading the necessary information. However, exceptions may occur while the file is still open causing the loss of information and exposing a weakness in our code. One clear way is to close a file using the `finally` block, after a `try` statement. A cleaner option is to use a *context manager* which is responsible for executing the `try` and `finally` statements and manage the life-cycle of the object in the context without the need to write these statements directly. The following code shows an example of the using a *context*:

```
# 25.py

with open('example_file_4', 'r', errors='replace') as file:
    content = file.read()
print(content)
```

```
file = open('example_file', 'r')
try:
    content = file.read()
finally:
    file.close()
```

If we execute `dir` in an `object` type, we can see that there are two methods called `__enter__` and `__exit__`:

```
# 26.py
```

```
2
3  file = open('example_file_4', 'w')
4  print(dir(file))
5  file.close()
```

```
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__',
'__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__iter__','__le__', '__lt__', '__ne__', '__new__', '__next__', '__reduce__',
'__reduce_ex__','__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_checkClosed', '_checkReadable', '_checkSeekable',
'_checkWritable', '_finalizing', 'buffer', 'close', 'closed', 'detach',
'encoding', 'errors', 'fileno', 'flush', 'isatty', 'line_buffering', 'mode',
'name', 'newlines', 'read', 'readable', 'readline', 'readlines', 'seek',
'seekable', 'tell', 'truncate', 'writable', 'write', 'writelines']
```

Both methods allow us to customize any object within a *context manager*. The `__exit__` method makes it possible to define the set of actions executed after a context finish. In the case of a file, it ensures that the context manager shall close the file correctly after reading the necessary data, even if an exception occurs while it is open.

In a similar way, the `__enter__` method let us specify the necessary steps performed to set the context of the object. For example, within a context the `open()` function returns a file object to the context manager. Finally, we simply use the `with` statement to generate the context and ensure that any object defined within it uses the `__enter__` and `__exit__` methods.

To personalize the use of any object within a *context manager*, we simply create a class and add the `__enter__` and `__exit__` methods. Then, call the class using the `with` statement. The following example shows how the `__exit__` method runs once we get out of the scope of the `with` statement.

```
1  # 27.py
2
3  import string
4  import random
5
6
7  class StringUpper(list):
8
```

```
9       def __enter__(self):
10          return self
11
12      def __exit__(self, type, value, tb):
13          for i in range(len(self)):
14              self[i] = self[i].upper()
15
16
17  with StringUpper() as s_upper:
18      for i in range(20):
19          # Here we randomly select a lower case ascii
20          s_upper.append(random.choice(string.ascii_lowercase))
21      print(s_upper)
22
23  print(s_upper)
```

```
['m', 'f', 'w', 'g', 'q', 'o', 'k', 'a', 'h', 'p', 'o', 'w', 'e', 'k', 'f',
't', 'e', 'n', 'm', 'l']
['M', 'F', 'W', 'G', 'Q', 'O', 'K', 'A', 'H', 'P', 'O', 'W', 'E', 'K', 'F',
'T', 'E', 'N', 'M', 'L']
```

In the last example we have a class that inherits from `list`. We implemented the `__enter__` and `__exit__` methods, hence we can instantiate it through a *context manager*. In this particular example, the *context manager* transforms all the lower case ascii characters to upper case.

10.2 Emulating files

We often have to interact with some software modules which only read and write data to and from files. In other cases, we just want to test a feature which requires some files. To avoid having to write data to persistent storage, we can have it on memory as files using `StringIO` or `BytesIO`. The next example shows the use of these modules:

```
1   # 28.py
2
3   from io import StringIO, BytesIO
4
5   # Simulate a text file
6   file_in = StringIO("info, text and more")
```

```
7  # Simulate a binary blob file
8  file_out = BytesIO()
9
10 char = file_in.read(1)
11 while char:
12     file_out.write(char.encode('ascii', 'ignore'))
13     char = file_in.read(1)
14
15 buffer_ = file_out.getvalue()
16 print(buffer_)
```

```
b'info, text and more'
```