Lecture 11.1 : Binary files

Introduction

- Our Python programs will often need to save their data to the hard disk. This is *persistent* storage i.e. data saved to the hard disk survives a reboot (unlike data stored in RAM). Our Python programs also need to be able to retrieve data from files on the hard disk. Until now our programs have worked exclusively with *text files*. Not all data takes the form of text, however. Consider images, audio recordings, movies, spreadsheets, etc. While text files are human-readable, *binary files* are not. If you open a binary file in a text editor it will look like gibberish!
- File processing entails the following steps:
    1. **Open the file:** This step initialises a *file object* that acts as a link from the program to the file on the disk. All subsequent file operations are invoked on the file object. (A file object is sometimes referred to as a *file descriptor* or *stream*.)
    2. **Read and/or write the file:** This is where the work is done. Through the file object the on-disk contents of the file will be read and/or written.
    3. **Close the file:** This step finalises the file and unlinks the file object from the program.

Opening and closing a binary file

- Below we `open()` a binary file called `image.png`. The first letter of the *mode* below is `r`. The `r` indicates the file has been opened only for reading (it cannot be written to or modified in any way). If the `open()` succeeds (why might it fail?) it returns a reference to a file object which we assign to the variable `f`. The second letter of the mode below is `b`.This indicates the file's type is *binary*. When finished with a file we `close()` it.

```
>>> f = open('image.png', 'rb')
>>> f.close()
```

- Here we opened the file for reading. Other modes in which a file can be opened include `w` for writing (warning: if the file already exists, when opened for writing it will be truncated i.e. its contents are deleted) and `a` for appending (additions to the file will follow any existing contents).

Reading a binary file

- We can read the entire contents of a binary file using the `read()` method:

```
>>> f = open('image.png', 'rb')
>>> content = f.read()
>>> len(content)
15411
```

- What type is `content`? If working with text files `content` would be a string. However, since we are working with a binary file, `content` is a *sequence type* named `bytes`:

```
>>> type(content)
<class 'bytes'>
```

- The drawback to `read()` is that it reads in *the entire file contents* and stores them in memory. If the file is particularly large this might not be the most efficient use of resources. An often better alternative is to process the file chunk-by-chunk.
- Previously, with text files, we could read in one line from the file at a time using the `readline()` method. Does it make sense to call `readline()` on a binary file? No. The `readline()` method reads everything up to the next newline character, `\n`. A binary file consists of a sequence of *arbitrary* bytes however. It is not composed of lines of text. Thus, in the binary file context, calling `readline()` makes no sense.
- Instead we continue to use the `read()` method but rather than reading in all of it we read a specific number of bytes from the file as follows:

```
>>> content = f.read(5)
>>> len(content)
5
```

- If we want to continue reading chunks until we reach the end of the file then we can do something like the following:

```
f = open('image.png', 'rb')
chunks = 0
size = 0
while True:
     content = f.read(10)
     if not content:
        break
     chunks += 1
     size += len(content)

print('Size: {}'.format(size))
print('Chunks: {}'.format(chunks))
```

- Above we simply keep reading from the file until we fail to read anything.
- Note that for the final chunk read, although we request 10 bytes, only 1 will be returned (since the file contains 15411 bytes). Thus it is important to check the quantity of data returned by a call to `read()` since the amount requested may not be available. Running the above program generates the following output:

```
$ python3 read_binary_file.py
Size: 15411
Chunks: 1542
```

Writing a binary file

- To write to a binary file we first open it for writing and then use the `write()` method to transfer data to it:

```
fin = open('image.png', 'rb')
fout = open('pic.png', 'wb')

while True:
     content = fin.read(10)
     if not content:
        break
     fout.write(content)

fin.close()
fout.close()
```

```
$ python3 copy_file.py
$ cmp image.png pic.png # The two files are identical
```

Changing position in a binary file

- When reading from a file Python tracks “where you are” in the file so that the next time you issue a read request, Python knows from where in the file to start taking the requested bytes.
- We can determine our current offset from the beginning of a file using the `tell()` method:

```
>>> f = open('image.png', 'rb')
>>> f.tell()
0
>>> content = f.read(10)
>>> content = f.read(10)
>>> f.tell()
20
```

- Suppose we wish to read the byte *N* from a file. With what we have covered so far we would be forced to read the preceding *N-1* bytes before we could read byte *N*. Our code might look something like this:

```
def read_byte_n_v01(filename, n):

    f = open(filename, 'rb')
    for i in range(n+1):
        content = f.read(1)

    print('Current offset: {}'.format(f.tell()))
    return content
```

- This above approach is not very efficient. Rather, if we wish to read only a specific range of bytes from a file we can move to a particular offset within the file and read the byte(s) directly (thereby *skipping* all preceding bytes). It is the `seek()` method that allows us to move to arbitrary offsets within the file. Using `seek()` our code becomes:

```
def read_byte_n_v02(filename, n):

    f = open(filename, 'rb')
    f.seek(n)
    content = f.read(1)
```

```
    print('Current offset: {}'.format(f.tell()))
    return content
```

- Let's verify it works:

```
def main():

    # Read byte number 1000 (indexing begins at 0)
    c1 = read_byte_n_v01('image.png', 999)
    print(c1[0])

    c2 = read_byte_n_v02('image.png', 999)
    print(c2[0])
```

```
$ python3 read_byte_n.py
Current offset: 1000
141
Current offset: 1000
141
```

- In addition to an offset the `seek()` method specifies an optional second parameter named `origin`. If `origin` is zero we move to *offset* bytes from the *beginning* of the file (the default behaviour). If `origin` is 1 we move forward *offset* bytes from the *current position*. If `origin` is 2 we to *offset* bytes relative to the *end* of the file.

- Thus to read the last byte in a file we can read in all of the file and look at the last byte as follows:

```
>>> f = open('image.png', 'rb')
>>> content = f.read()
>>> content[-1]
130
```

- Or, we can move to the byte at offset `end-of-file-1` and read a single byte from there:

```
>>> f = open('image.png', 'rb')
>>> f.seek(-1, 2)
15410
>>> content = f.read(1)
>>> content[0]
130
```

Serializing objects

- Consider the following Python excerpt which creates a `list` object called `mylist`:

```
>>> import random
>>> mylist = random.sample(range(-99999, 100000), 100000)
>>> isinstance(mylist, list)
True
```

- Supposing we want to save `mylist` so that another program can use it in the future. How might we do that? We could write out the contents of `mylist` to a text file from which our program could

rebuild its own version of `mylist`. Our program would have to read in the sequence of numbers (as strings) from the text file and add them to its list (having converted each from a string to an integer). This would be a rather painful process. Is there any easier way?

- Furthermore, unlike the list above, certain objects do not have an obvious textual representation. Consider the following Python program which creates several objects:

```
from employee_091 import Employee, Manager, AssemblyWorker

def main():

    e1 = Manager('Mary', 1, 50000)
    e2 = AssemblyWorker('Fred', 2, 15.50, 40)
    e3 = Employee('Sean', 3)

    print(e1)
    print(e2)
    print(e3)

if __name__ == '__main__':
    main()
```

How can we save the `e1`, `e2` and `e3` objects to a file so that they can be used again in the future?

- Converting an object to a stream of bytes so it can be simply saved to (and retrieved from) a file *as an object* is called *serializing* an object. Python provides a module for serializing objects. That module is called `pickle`.
- After importing the `pickle` module we perform the following steps to pickle an object and save it to a file:
    - Open a binary file for writing,
    - Call the `pickle` module's `dump()` method to pickle (serialize) the object and write it to the file,
    - Once all objects have been saved to the file, close it.
- Here is an example of pickling an object and saving it to a file:

```
import pickle
import random

def main():

    mylist = random.sample(range(-99999, 100000), 100000)

    f = open('pickled.dat', 'wb')
    pickle.dump(mylist, f)
    f.close()

if __name__ == '__main__':
    main()
```

- To retrieve a pickled object from a file we use the `load()` method. Here is an example:

```
import pickle
import random

def main():
```

```
    mylist = random.sample(range(-99999, 100000), 100000)

    f = open('pickled.dat', 'wb')
    pickle.dump(mylist, f)
    f.close()

    f = open('pickled.dat', 'rb')
    anotherlist = pickle.load(f)
    f.close()

    print(mylist == anotherlist)

if __name__ == '__main__':
    main()
```

```
$ python3 pickle_list.py
True
```

- What about pickling and restoring a number of objects?

```
from employee_91 import Employee, Manager, AssemblyWorker
import pickle

def main():

    e1 = Manager('Mary', 1, 50000)
    e2 = AssemblyWorker('Fred', 2, 15.50, 40)
    e3 = Employee('Sean', 3)

    f = open('pickled_employees.dat', 'wb')
    pickle.dump(e1, f)
    pickle.dump(e2, f)
    pickle.dump(e3, f)
    f.close()

    f = open('pickled_employees.dat', 'rb')
    emp1 = pickle.load(f)
    emp2 = pickle.load(f)
    emp3 = pickle.load(f)
    f.close()

    print(emp1)
    print(emp2)
    print(emp3)

if __name__ == '__main__':
    main()
```

```
$ python3 pickle_employees.py
Name: Mary
Number: 1
Wages: 961.54
Name: Fred
Number: 2
Wages: 620.00
Name: Sean
Number: 3
Wages: 0.00
```

- When we restore an instance of a user-defined class the module that defines the class must be available. It is automatically imported so that Python can make sense of the restored object. If the

module is not available you will see an error like the following:

```
>>> import pickle
>>> f = open('pickled_employees.dat', 'rb')
>>> e1 = pickle.load(f)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'employee_91'
```