Section B2 Files & Data

Files

Opening a File

In Python, files are opened using the built-in **open()** function. The call to open() returns a *file object*.

- The first, and only mandatory, parameter for open() is the file name. The function above works only if your Python code and your file are in the same directory. If they are in different directories, you will need to provide the 'full path' to the file.
- The second parameter is the file's *mode*. This is an optional string that specifies the mode the file is opened with. If you do not set a mode, the default mode **'r'** is used to allow read-only access.

Table 1. Summary of the behaviour of the four most common modes.

Description	mode	open for reading?	open for writing?	Behaviour
read-only	`'r'`	Yes	No	Fails if the file does not exist.
write-only	`'w'`	No	Yes	If the file already exists, the file's contents is truncated (cleared), otherwise it will create an empty file.
exclusive creation	`'x'`	No	Yes	Creates an empty file; fails if the file already exists.
append-only	`'a'`	No	Yes	If the file already exists, all writes are appended to the end of it, otherwise it will create an empty file.

A **'+'** can be appended to each of the above modes to extend their permissions to both read and write. For example open('file0.txt', 'r+') will extend the 'read-only' functionality to 'read-and-write', which will also allow you to write new text to the file using write().

```
with open('file0.txt', 'r+') as f:
    # do something
```

when we use with-as, the file is automatically closed when the block completes or an exception is thrown.
Remember, if you did not use the with-as approach, you need to manually close the file using close()**.**

Reading from a File

Firstly, you can read either the entire file or a specific number of characters using the **read()** method.

```
myFile = open('file0.txt')
print(myFile.read(5))
myFile.close()
```

```
hello
```

Secondly, you can read the file line-by-line using
the readline() method. Regardless of the size is specified, the read will stop once the line ends or the number of bytes specified in the call
to read() or readline() has been reached.

```
myFile = open('file0.txt')
print(myFile.readline())
print(myFile.readline(4))
myFile.close()
```

```
hello world

nice
```

If you want to return to a place that you have already read you can use **seek()** method and provide it with the byte (or character) you want to return to. As such seek(0) returns to the top of the file.
Why is there a blank line between "world" and "nice" in above?
myFile.readline() is returning a string that contains the first line of the file. At the end of that line is a newline character to signify the line has ended. This character will be included in the string generated by readline().
The print() function also places a newline after it has run. Therefore, when print(myFile.readline()) is run, there are two newlines being output to the terminal, resulting in the blank line being shown.

Writing to a File
Writing to a file is achieved with the **write()** function by giving it a string. All writes are appended to the end of a file.

```
myFile = open('file0.txt', 'w')
myFile.write('goodbye world')
myFile.close()
```

```
goodbye world
```

```
myFile = open('file0.txt', 'a')
myFile.write('goodbye world')
myFile.close()
```

```
hello world
nice to meet yougoodbye world
```

```
myFile = open('file0.txt', 'x')
myFile.write('goodbye world')
myFile.close()
```

```
---------------------------------------------------------------------------
FileExistsError                           Traceback (most recent call last)
/tmp/ipykernel_4278/530682198.py in
----> 1 myFile = open('file0.txt', 'x')
      2 myFile.write('goodbye world')
      3 myFile.close()

FileExistsError: [Errno 17] File exists: 'file0.txt'
```

When you open a file in 'x' mode (exclusive creation mode), Python tries to create and open that file. But if the file already exists, instead of overwriting it, Python throws a FileExistsError. This is to ensure you don't accidentally overwrite content in an existing file.

Handling Structured Data (CSV, JSON)

Working with CSV files

CSV (comma-separated values) are a common form of text file. The comma is called a "delimiter" and there are many cousins of CSV files that use, for example, tab characters as a delimiter. Headers are not mandatory in CSV files but, if you have the option, you should always include a header row in your files to make them self documenting.

Reading a CSV file

Python has a CSV module that exists to make reading from CSV files easier. The CSV module defines **readers** which allow you to read each row. There are two options:

1. reader: reads a row and returns the values as a *list* of strings
2. DictReader: reads each row and return the values as an *ordered dictionary* where the first row is used as the keys.

```
import csv
dataFile =  open('customers.csv', 'r')

reader = csv.reader(dataFile)

for row in reader:
    print(row)

dataFile.close()
```

```
['name', 'age', 'postcode']
['John', '52', '5002']
['Ye', '18', '3005']
['Siobhan', '34', '2356']
```

Writing to a CSV file

Just like with reading, the CSV module offers two options to make writing easier.

1. writer: writes comma delimited values to a file
2. DictWriter: writes the keys as comma delimited header values and the dictionary values as comma delimited row values

```
import csv

data = [['name','age','postcode'],
    ['John', 52, 5002],
    ['Ye', 18, 3005],
    ['Siobhan', 34, 2356]]
extra_data = ['Jose', 44, 5125]

with open('output.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerows(data)
    writer.writerow(extra_data)
```

The reason we used **writerows** (plural) the first time and **writerow** (singular) the second is because the first time we are adding a list of lists which represents multiple rows of data.

Working with JSON files

The JSON format is built on two structures:

- A collection of name/value pairs. In Python, this is realized as a dictionary.
- An ordered list of values. In Python, this is realized as a list.

A JSON object is an unordered set of name/value pairs grouped by *curly braces*: {}

Working with JSON in Python is made significantly easier through use of

the JSON module
. This module handles the task of writing Python data to file (called "serialization") and reading from a JSON file or string into a Python data structure (called "deserialisation").
During the process of serialisation and deserialisation, simple Python objects are translated to and from JSON according to a fairly intuitive conversion

Python	JSON
dict	object
list, tuple	array
str	string
int, long, float	number
True	true
False	false
None	null

Deserialisation - Reading JSON Data
json.loads() is given a string in JSON data format and loads the data into dictionaries (for name/value pairs) and lists (for unlabelled sequences).

Reading a JSON file
To read a JSON file:

1. Open the file. The file does not need to be a JSON file, but the entire contents does need to be in JSON format.
2. Read from the file using read(). This will return the entire file contents as a string in JSON format.
3. Call the json.loads() on the string generated in Step 2.

```
import json

with open('customers.json', encoding='utf-8-sig') as dataFile:
    # load data into dictionary
    jstr = dataFile.read()
    jdata=json.loads(jstr)

print(jdata)
```

When working with JSON, remember that name:value pairs in JSON will become dictionary elements, and comma separated values within [ ] will become part of a list.

Serialisation - Writing JSON Data
Simply open the file for writing '**w**' or appending '**a**' and then call **json.dump(data, file)**.

```
import json

data = [{'name':'John', 'age':52, 'postcode':5002},
        {'name':'Ye','age':18, 'postcode':3005},
        {'name':'Siobhan', 'age':34, 'postcode':2356}
       ]

with open('newFile.json','w') as file:
    json.dump(data,file)
```

Handling Structured Data (XML)

Reading and parsing an XML file

Reading an XML document creates a Document Object Model (DOM) from which we can access elements in the doc. This process is sharply different from reading and manipulating a JSON file in that the DOM is a more complex object representation queried using the id's and attributes of elements.

```
from xml.dom.minidom import parse

# Parse XML from a file object
with open("face.svg") as file:
     document = parse(file)

print (f"version: {document.version}, encoding: {document.encoding}.")

ellipse_elements = document.getElementsByTagName("ellipse")
for eye in ellipse_elements:
    colour = eye.getAttribute("fill")
    width =  eye.getAttribute("rx")
    height =  eye.getAttribute("ry")
    print(f"eye colour: {colour}, width: {width}, height: {height}")
```

```
version: 1.0, encoding: UTF-8.
eye colour: black, width: 6, height: 8
eye colour: black, width: 6, height: 8
```

Modifying the DOM

We can also modify the DOM in memory and then write back the file as a new XML document.

```
for eye in ellipse_elements:
    eye.setAttribute("fill", "red")
    eye.setAttribute("rx", "10")
    eye.setAttribute("ry", "10")

with open('red-eye.svg', 'w') as out_file:
    document.writexml(out_file, indent = "  ", addindent = "  ", newl = "\n")
```