FILE HANDLING

DR IAN CORNELIUS

HELLO

- Learning Objectives
    1. Understand how to handle multiple file types in Python
    2. Demonstrate the ability to handle files of different types

INTRODUCTION TO FILES

- Files are an object on a computer that stores either data, information, settings or commands
- They are great for:
  - storing data permanently
  - sharing of data between applications
  - storing a huge amount of data

TYPES OF FILES

- There are two file types:
  - **text**: this is where data is stored in the form of strings
    - i.e. `.txt`, `.py`, `.cpp`
  - **binary**: this is where data is stored in the form of bytes
    - i.e. `.jpg`, `.gif`, `.png`, `.exe`

OPENING A FILE (1)

- Files can be opened in Python using the `open()` function
- There are various opening modes:

Mode	Definition
`w`	writes data, if the file already exists then the data will be lost
`r`	reads data, the cursor inside the file is positioned at the beginning
`w+`	writes and reads data, previous data in the file will be lost
`r+`	reads and writes data, previous data in the file will not be deleted, and the cursor inside the file is positioned at the beginning
`a+`	appends and reads data, the file cursor is positioned at the end of the file
`x`	creates the specified file, or returns an error if the file exists

OPENING A FILE (2)

- Files can also be specified if they should be handled in a binary or text mode using:
  - `t` - for text mode (the default option)
  - `b` - for binary mode

```
exampleFile = open("filename.extension", "w")
```

CLOSING A FILE

- Files are closed in Python using the `close()` function
- You must always call this function when you have finished using or processing a file

```
exampleFile = open("myfile.txt", "w")
exampleFile.write("Hello 4061CEM")
exampleFile.close()
```

READING CONTENTS FROM A FILE

- Reading the contents of a file can be achieved in one of three ways:
  - `read()`: will read all the lines and return them line-by-line
  - `read(n)`: will read n bytes from the file
  - `readlines()`: will return all strings as elements in a list

```
exampleFile = open("myfile.txt")
strRead = exampleFile.read()
print(strRead)
exampleFile.close()
```

```
Traceback (most recent call last):
  File "source.py", line 1, in <module>
    exampleFile = open("myfile.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'myfile.txt'
```

SETTING THE FILE CURSOR POSITION

- To read from a set position in a file, this can be achieved by using the `seek()` function
  - the function will also return the new position once it has finished reading the line

```
exampleFile = open("myfile.txt")
exampleFile.seek(4)
print(exampleFile.read())
exampleFile.close()
```

WRITING TO AN EXISTING FILE

- Writing content to an existing file can be used with the `open()` function and one of the following modes:
  - `a` for append mode, which will append contents to the end of the file
  - `w` for write mode, which will overwrite existing content

```
exampleFile = open("myfile.txt", "w")
exampleFile.write("Oops, I overwrote the content of the file.\n")
exampleFile.close()
```

USING THE `WITH` KEYWORD

- Reading a file can be achieved using the `with` keyword
  - a benefit is that it will take care of closing the file automatically
  - therefore, no requirement to call the `close()` function

```
with open('myfile.txt', 'w') as exampleFile:
        exampleFile.write("Hello 4061CEM\n")
        exampleFile.write("This is an exciting module.\n")
```

```
with open('myfile.txt', 'r') as exampleFile:
        for line in exampleFile:
            print(line)
```

WORKING WITH DIRECTORIES (1)

- Working with directories and other files can be obtained by using the `os` module

CHECKING THE WORKING/ACTIVE DIRECTORY

- To determine the directory you are currently working inside can be achieved using the `getcwd()` function

```
import os
currentDir = os.getcwd()
print(f"Current working directory is: {currentDir}")
```

WORKING WITH DIRECTORIES (2)

CREATING A DIRECTORY

- Creating a directory can be achieved using the `mkdir()` function

```
import os
os.mkdir("my_directory")
```

- Using a forwards slash (“`/`”) in the string will create a subdirectory
  - you must ensure that the top-level directory has already been created (“`my_directory`”)

```
os.mkdir("my_directory/sub_directory")
```

WORKING WITH DIRECTORIES (3)

DELETING A DIRECTORY

- A directory can be deleted by using the `rmdir()` function
  - only empty directories can be deleted

```
import os
os.rmdir("my_directory")
```

WORKING WITH DIRECTORIES (4)

RENAMING A DIRECTORY

- The renaming of a directory can be achieved using the `rename()` function

```
import os
os.rename("my_directory", "a_new_name_directory")
```

CHECKING IF A FILE EXISTS

- To check whether a file exists can be achieved using the `isfile()` function

```
import os
fileName = input("Enter a Filename:")
if os.path.isfile(fileName):
    print("File Exists")
else:
    print("File does not exist")
```

DELETING A FILE

- Deleting a file can be achieved using the `remove()` function

```
import os
os.remove("myfile.txt")
```

DISPLAYING ALL CONTENTS OF A FOLDER

- To determine all folders and files of a directory can be achieved using the `walk()` function

```
import os
    for dirPath, dirNames, filenames in os.walk('/'):
        print(f"Current Path: {dirPath}")
        print(f"Directories: {dirNames}")
        print(f"Files: {filenames}\n")
```

RUNNING EXECUTABLES

- The `os` module also contains a `system()` function which is useful to run shell commands from within Python

```
import os
os.system('dir')
os.system('python3 sample_script.py')
```

HANDLING AND USING CSV FILES

- `csv` is an abbreviation for **C**omma-**S**eparated **V**alues
- The file extension for these types of files are `.csv`
- Commonly used to store plain-text data
- Uses a comma (“`,`”) to separate the values within a file, hence the name
  - however, other characters can be used; such as a semi-colon (“`;`”)
- Theory behind CSVs is to enable the exporting of data to a universal file type
  - then be able to import the data back into an application

STRUCTURE OF A CSV FILE (1)

- The first line of the file is considered to be the label, header or column name row
  - these values are often used to refer to the data for a particular column
- An example layout of a `csv` file can be something such as:

```
name,age,course
Ian,33,Computer Science
Terry,Unknown,Computer Science
```

- The first line consisting of: `name, age, course` are the labels for each column of data
- Subsequent lines following this are the data, with a comma (“`,`”) separating each value for the columns

STRUCTURE OF A CSV FILE (2)

- You may picture the contents of a `csv` file as a table, for example:

name	age	course
Ian	33	Computer Science
Terry	Unknown	Computer Science

READING A CSV FILE IN PYTHON (1)

- There are two methods of reading a CSV file:
  - csv
  - DictReader

CSV

- The `csv` module enables you to read the contents of a file
- Reads each line of the file and stores the data as a list
  - each item in the row being an item in the list
- Not the most functional way of reading the contents of a CSV file
  - does not utilise the labels or column headers of the file

DICTREADER

- An alternative method of reading a CSV file is using `DictReader` from the `csv` module
- Maps the content of the CSV file to a dictionary data type
- Provides a more useful method of importing data
  - you can read the whole CSV file and only access elements you want

READING A CSV FILE IN PYTHON (2)

METHOD 1 - csv EXAMPLE

```
import csv
with open("data.csv", "r") as csvFile:
    csvReader = csv.reader(csvFile, delimiter=",")
    for row in csvReader:
        print("\n\n", row)
```

READING A CSV FILE IN PYTHON (3)

METHOD 2 - `DICTREADER` EXAMPLE I

```
import csv
with open("data.csv", "r") as csvFile:
    csvReader = csv.DictReader(csvFile)
    for row in csvReader:
        print("\n\n", row)
```

READING A CSV FILE IN PYTHON (4)

METHOD 2 - `DICTREADER` EXAMPLE II

- If you want to capture only the names in the file, you can adapt the code to be akin to:

```
import csv
names = []
with open("data.csv", "r") as csvFile:
    csvReader = csv.DictReader(csvFile)
    for row in csvReader:
        names.append(row["name"])
print("\n\n", names)
```

WRITING TO A CSV FILE IN PYTHON (1)

- There are two methods of creating a CSV file:
  - `csv`
  - `DictWriter`

CSV

- The `csv` module also enables users to write to a CSV file
  - i.e. you want to add a new row to the contents in the file already

DICTWRITER

- An alternative method to write to a file is the `DictWriter` class
- Create a new dictionary object that has the details that you want to be written to the file
  - good practice is to use the same labels/keys as those used in the `csv` file
- Then you can easily write the data to the file by just calling the dictionary object in `writerow()`

WRITING TO A CSV FILE IN PYTHON (2)

METHOD 1 - csv EXAMPLE

```
import csv
with open("data.csv", "a", newline='\n') as csvFile:
    csvWriter = csv.writer(csvFile, delimiter=",")
    csvWriter.writerow(['Daniel', 'Unknown', 'Ethical Hacking and Cyber Security'])
```

```
name,age,course
Ian,33,Computer Science
Terry,Unknown,Computer Science
Daniel,Unknown,Ethical Hacking and Cyber Security
```

WRITING TO A CSV FILE IN PYTHON (3)

METHOD 2 - `DICTWRITER` EXAMPLE

```
</> import csv
    aStudent = {"name": "Daniel",
                "age": "Unknown",
                "course": "Ethical Hacking and Cyber Security"}
    with open("data.csv", "a", newline='\n') as csvFile:
        csvWriter = csv.DictWriter(csvFile, fieldnames=aStudent.keys())
        csvWriter.writerow(aStudent)
```

```
name,age,course
Ian,33,Computer Science
Terry,Unknown,Computer Science
Daniel,Unknown,Ethical Hacking and Cyber Security
```

GOODBYE

- Questions?
  - Post them in the **Community Page** on Aula
- Contact Details:
  - Dr Ian Cornelius, ab6459@coventry.ac.uk