Chapter 10: Python Files

Contents

Chapter 10: Python Files 1
DRY 2
Code Shape 2
Temporary Memory 2
Secondary Storage 3
Files 3
  Python File Modes 5
  Opening and Reading a Text File 5
  Directories 6
Tutorial 10.1: Reading a Text File 6
  Using the with Statement 7
  Writing to a Text File 8
Tutorial 10.2: Writing a Text File 9
Tutorial 10.3: Reading a Text File Line by Line 10
  rstrip String Method 11
Tutorial 10.4: Appending to a Text File 12
File Processing Cheat Sheet 13
Tutorial 10.5: Writing Numbers to a Text File 13
Tutorial 10.6: Read Numbers from a Text File 15
Tutorial 10.7: Reading and Writing to a Text File 15
JSON 18
Tutorial 10.8: Dumping and Loading a JSON File 19
    Dump JSON to File 19
    Load JSON Data 20
Text Files and Binary Files 22
  Reading a Binary File 23
Assignment 1: Serialize Yourself Baby 23
  Glossary 23

Assignment Submission.....24

 No AI use.

Time required: 120 minutes

DRY

Don't **R**epeat **Y**ourself

Code Shape

Please group program code as follows.

- Declare constants and variables
- Get input
- Calculate
- Display

Temporary Memory

Temporary memory, often referred to as "volatile memory," is a type of computer memory that holds data temporarily while the system is running. It is crucial for storing data that the CPU needs immediate access to during tasks. It includes Random Access Memory (RAM) and Cache memory.

RAM: This volatile memory type allows rapid reading and writing of data, facilitating quick access for the CPU. It holds data that applications are actively using or that the operating system requires. Once the computer is turned off or restarted, the data stored in RAM is erased.

Cache Memory: This high-speed memory resides closer to the CPU, serving as a buffer between the processor and the slower main memory (RAM). It stores frequently accessed data and instructions, allowing the CPU to retrieve them quickly. Cache memory is also volatile and gets cleared when the computer is powered down.

Secondary Storage

Secondary storage refers to non-volatile, long-term storage devices used to retain data even when a computer system is powered off. Unlike primary or temporary memory (e.g., RAM), secondary storage devices store data persistently for future retrieval and long-term use. Common examples of secondary storage include hard disk drives (HDDs), solid-state drives (SSDs), optical discs (like CDs and DVDs), and USB flash drives.

1. **Non-volatile**: Unlike temporary memory (RAM), secondary storage retains data even when the computer is turned off or restarted.
2. **Capacity**: Secondary storage devices typically offer larger storage capacities compared to volatile memory, allowing users to store large amounts of data for an extended period.
3. **Slower access speed**: Retrieving data from secondary storage tends to be slower compared to primary memory like RAM due to mechanical or electronic limitations inherent in the storage devices.
4. **Persistent storage**: It is used to store operating systems, software applications, personal files, multimedia content, and other data that don't need to be accessed as frequently as the data stored in primary memory.

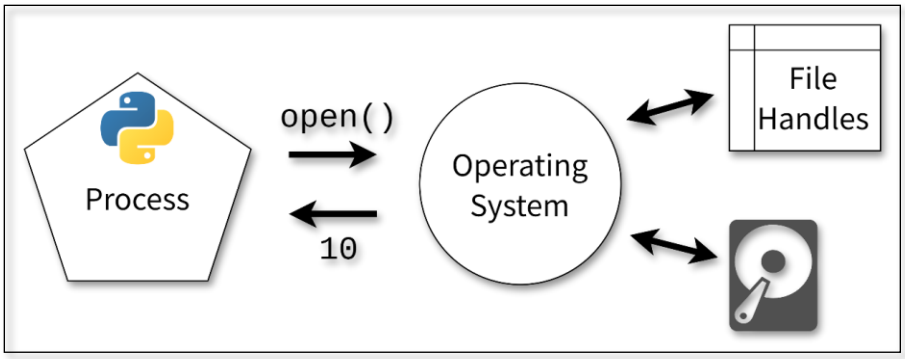
Secondary storage devices serve as the primary means for data archiving, backup, and long-term retention, complementing the temporary nature of primary memory in computer systems.

Files

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output. When they end, their data disappears. If you run the program again, it starts with a clean slate.

Python delegates file operations to the operating system. The operating system is the mediator between processes, such as Python, and all the system resources, such as the hard drive, RAM, and CPU time.

When you open a file with **open()**, you make a system call to the operating system to locate that file on the hard drive and prepare it for reading or writing. The operating system will then return an unsigned integer called a **file handle** on Windows and a **file descriptor** on UNIX-like systems, including Linux and macOS:



Operating systems limit the number of open files any single process can have. This number is typically in the thousands. Operating systems set this limit because if a process tries to open thousands of file descriptors, something is probably wrong with the process. Even though thousands of files may seem like a lot, it's still possible to run into the limit.

Apart from the risk of running into the limit, keeping files open leaves you vulnerable to losing data. In general, Python and the operating system work hard to protect you from data loss. If your program or computer crashes, the usual routines may not take place, and open files can get corrupted.

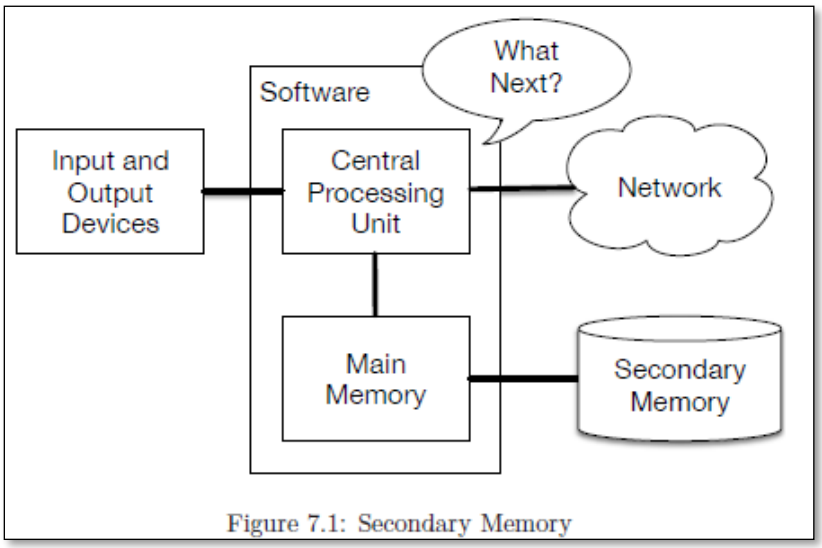


Figure 7.1: Secondary Memory

Secondary memory or storage is not erased when the power is turned off. In the case of a USB flash drive, the data we write from our programs can be removed from the system and transported to another system.

We will primarily focus on reading and writing text files such as those we create in a text editor.

Python File Modes

r	Opens a file for reading only. (default)
w	Opens a file for writing. Creates a new file if it does not exist or erases the contents of the file if it exists.`
a	Opens a file for appending at the end of the file without erasing the contents. Creates a new file if it does not exist.

Opening and Reading a Text File

When we want to read or write a file (say on your hard drive), we first must open the file. Opening the file communicates with your operating system, which knows where the data for each file is stored. When you open a file, you are asking the operating system to find the file by name and make sure the file exists.

In this example, we open the file example.txt, which should be stored in the same folder that you are in when you start Python.

```
# Open a file in the same folder as the program
# We create an object named text_file
# This is called a file handle
text_file = open('example.txt', 'r')
```

If the open is successful, the operating system returns a file handle. The file handle is not the actual data contained in the file, but instead it is a "handle" that we can use to read the data. You are given a handle if the requested file exists, and you have the proper permissions to read the file.

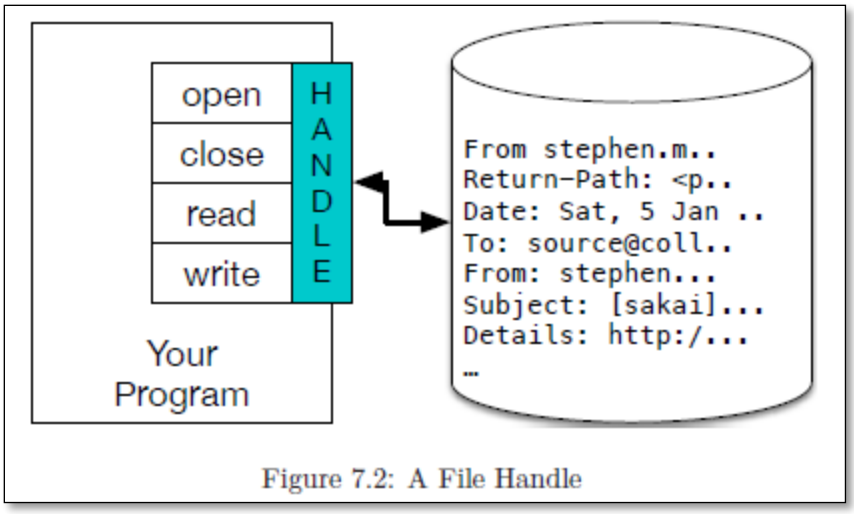


Figure 7.2: A File Handle

The string s is now the following.

```
'Hello.\nThis is a text file.\nBye'
```

If the file does not exist, open will fail with a traceback and you will not get a handle to access the contents of the file.

Directories

Say your program opens a file, like below:

```
s = open('file.txt').read()
```

The file is assumed to be in the same directory as your program itself. If it is in a different directory, then you need to specify the full path to the file, like below:

```
s = open('c:users/loring/desktop/file.txt','r')
```

Tutorial 10.1: Reading a Text File

r	Opens a file for reading only. (default)

In the same folder, create a txt file named **example.txt** Include the following text.

```
Hello
This is a sample text file.
Bye!
```

Create and test the program as listed. Name it **file_read.py**

```
"""
    Name: file_read.py
    Author:
    Created:
    Purpose: Read and display a text file
"""

# Open a file in the same folder as the program
# We create an file object named file_handle
file_handle = open("example.txt", "r")

# Read the entire contents of the file into a string
contents = file_handle.read()

# Close the file handle
file_handle.close()

# Print the string
print(contents)
```

In Windows, the file object created is called a file handle. The variable name **file_handle** is used to help understand the operation.

Using the with Statement

We have explicitly opened and closed a file. This is to help understand how to work with files. You can also let Python do all the work.

You must close the file each time your program finishes. Ensuring that the file is opened and closed correctly is good file management as it reduces the risk of data loss. This is good practice and Python has built-in functions to simplify this.

Using a **with** statement you can open a file, do whatever you want with it, and then automatically close it.

The **with** statement also provides a means to manage any errors. For example if the file is not present, or the file name is incorrect.

This example program will open a file, store it in the variable **file_handle**, read the contents of the file to the variable **data**, then print it to the screen.

```
with open("test.txt") as file_handle:
    data = file_handle.read()
    print(data)
```

Note: No code is needed to close the file, as the **with** statement handles closing the file once the contents have been read.

```
"""
    Name: file_read_with.py
    Author:
    Created:
    Purpose: Read and display a text file using the 'with' operator
"""


# Open a file in the same folder as the program
# The 'with' operator creates a file object named file_handle
with open("example.txt") as file_handle:

    # Read the entire contents of the file into a string
    contents = file_handle.read()

# File handle closes automatically when you exit the with statement
# Print the string
print(contents)
```

1. The **with** statement above opens the file with a file handle variable: **file_handle**
2. Inside the **with** statement the file is open. You can perform any file operation inside the with statement.
3. Python closes the file when the **with** statement exits.

Writing to a Text File

w	Opens a file for writing. Creates a new file if it does not exist or erases the contents of the file if it exists.

Let's write to a file called **rockstars.txt**

```
# Create a file handle
# for writing to rockstars.txt
rock_file = open('rockstars.txt', 'w')

# Write the names of three rock stars to the file
# Substitute your favorites
# \n is an escape character creating a new line
rock_file.write(f'Eddie Van Halen\n')
rock_file.write(f'Eric Clapton\n')
rock_file.write(f'Stevie Nicks\n')
```

The first line opens the file rockstars.txt with a file handle named **rock_file**. The 'w' indicates that we want to write to the file.

To write to the file, we use the **.write()** method with F-strings formatting. When we are done writing, we should close the file to make sure our changes take. Be careful here because if writefile.txt already exists, its contents will be overwritten.

Tutorial 10.2: Writing a Text File

Create and test the following Python program called **file_rockstars.py**

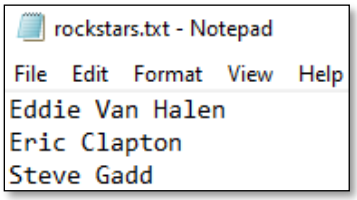
This program uses try: except: to provide error handling. If there was an exception, inform the user.

Let the user know the operation was successful by including a print statement in the try block.

```python
"""
    Name: file_rockstars.py
    Author:
    Created:
    Purpose: Write 3 lines of data to a text file
"""

# Catch any exceptions
try:
    # Create a file handle for writing to rockstars.txt
    with open("rockstars.txt", "w") as file_handle:

        # Write the names of three rock stars to the file
        # Substitute your favorite artists
        # \n is an escape character creating a new line
        file_handle.write(f"Eddie Van Halen\n")
        file_handle.write(f"Eric Clapton\n")
        file_handle.write(f"Steve Gadd\n")

    print(f"File written successfully.")

# Let the user know if there was an exception
except Exception as e:
    print(f"The file was not written: {e}")
```

Open **rockstars.txt** to check your work.

Example run:



Tutorial 10.3: Reading a Text File Line by Line

Add this code to **file_rockstars.py**

This program reads each line into a separate string.

```
# Catch any exceptions in the program
try:
    # Open a file in the same folder as the program
    with open('rockstars.txt', 'r') as text_file:

        # Read each line into a separate string
        line1 = text_file.readline()
        line2 = text_file.readline()
        line3 = text_file.readline()
    # File handle automatically closes

    # Print the strings
    print(line1)
    print(line2)
    print(line3)
    print('The file was successfully read.')

# Let the user know if there was an exception
except Exception as e:
    print(f'There was a problem reading the file. {e}')
```

Because of the **\n** newline character we added when we wrote the file, there is a space between each line when we print out the strings.

Example run:

```
Eddie Van Halen

Eric Clapton

Steve Gadd

The file was successfully read.
```

rstrip String Method

We will use the **rstrip()** text method to remove **\n** from the right side of the string. Add the new code to **file_rockstars_read.py**

```python
# Catch any exceptions in the program
try:
    # Open a file in the same folder as the program
    with open('rockstars.txt', 'r') as file_handle:

        # Read each line into a separate string
        line1 = file_handle.readline()
        line2 = file_handle.readline()
        line3 = file_handle.readline()

    # Strip \n from each string with the rstrip function
    line1 = line1.rstrip('\n')
    line2 = line2.rstrip('\n')
    line3 = line3.rstrip('\n')

    # Print the strings
    print(line1)
    print(line2)
    print(line3)

    print('The file was successfully read.')

# Let the user know if there was an exception
except Exception as e:
    print(f'There was a problem reading the file. {e}')
```

The text file displayed without \n extra new line characters.

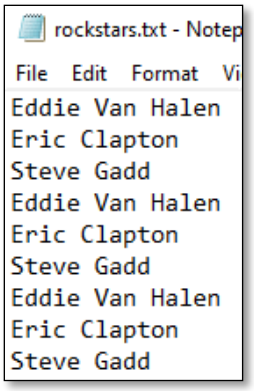
```
Eddie Van Halen
Eric Clapton
Steve Gadd
The file was successfully read.
```

Tutorial 10.4: Appending to a Text File

a	Opens a file for appending at the end of the file without erasing the contents. Creates a new file if it does not exist.

Open **file_rockstars.py**. The only change needed is to change the 'w' to an 'a'. Run the program a couple times. Open the text file to make sure the program worked.

Example run:



File Processing Cheat Sheet

You can **read** an existing file with Python:

```
with open("file.txt", "r") as file:
    content = file.read()
```

You can **create** a new file with Python and **write** some text on it:

```
with open("file.txt", "w") as file:
    content = file.write("Sample text")
```

You can **append** text to an existing file without overwriting it:

```
with open("file.txt", "a") as file:
    content = file.write("More sample text")
```

You can both **append and read** a file with:

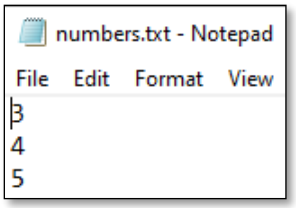
```
with open("file.txt", "a+") as file:
    content = file.write("Even more sample text")
    file.seek(0)
    content = file.read()
```

Tutorial 10.5: Writing Numbers to a Text File

Numbers must be converted to a string before they can be written to a text file. We will use f-strings to convert the numbers to strings.

```python
"""
    Name: file_numbers_write.py
    Author:
    Created:
    Purpose: Numbers must be converted to strings before they
             are written to a text file.
"""
# Constant for filename
FILE_NAME = "numbers.txt"

# Catch any exceptions
try:
    # Open a file for writing
    with open(FILE_NAME, "w") as file_handle:

        # Get three numbers from the user
        number1 = int(input("Enter a whole number: "))
        number2 = int(input("Enter another whole number: "))
        number3 = int(input("Enter another whole number: "))

        # Write the numbers to the file using f-strings
        file_handle.write(f"{number1}\n")
        file_handle.write(f"{number2}\n")
        file_handle.write(f"{number3}\n")

        # Let the user know it worked
        print("Data was written to numbers.txt")

# Let the user know there was trouble
except Exception as e:
    print(f"There was trouble writing to the file. {e}")
```

Example run:



Tutorial 10.6: Read Numbers from a Text File

Numbers stored as text in a text file must be converted from strings to numbers before they can be used in calculations.

```
"""
    Name: file_read_numbers.py
    Author:
    Created:
    Purpose: Numbers must be converted from strings to ints
"""
# Constant for filename
FILE_NAME = "numbers.txt"

# Catch any exceptions
try:
    # Open a file for reading
    with open(FILE_NAME, "r") as number_file:
        # Read 3 numbers from a file
        number1 = int(number_file.readline())
        number2 = int(number_file.readline())
        number3 = int(number_file.readline())

    # Sum the numbers
    total = number1 + number2 + number3

    # Display the numbers and the total
    print(f"The numbers are: {number1} {number2} {number3}")
    print(f"The total is: {total}")

# Let the user know there was trouble
except Exception as e:
    print(f"There was trouble reading the file. {e}")
```

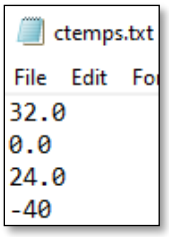
Example run:

```
The numbers are: 3 4 5
The total is: 12
```

Tutorial 10.7: Reading and Writing to a Text File

We will write a program that reads a list of temperatures from a file called **ctemps.txt**, converts those temperatures to Fahrenheit, and writes the results to a file called **ftemps.txt**.

Create a text file named **ctemps.txt** in the current folder and put the following numbers in it. Don't put any extra line returns in it, stop typing at -40.

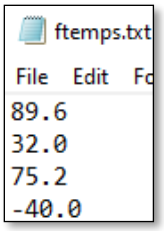


Create and test the following program named **convert_temperatures.py**

```python
"""
    Name: file_temperatures.py
    Author:
    Created:
    Purpose: Read a list of temperatures from a file,
    convert them to Fahrenheit
    Write the results to a file and to the screen
"""
# Constant for filename
C_FILE = "ctemps.txt"
F_FILE = "ftemps.txt"

# Catch any exceptions
try:
    # Open a file for writing
    with open(C_FILE, "r") as c_file, open(F_FILE, "w") as f_file:
        # Strip the spaces off of every line in the file
        # using a list comphrehension
        c_temp = [line.rstrip() for line in c_file]

        # Loop until all the temperatures have been converted
        for temperature in c_temp:
            # Convert the temperature to Fahrenheit
            f_temp = float(temperature) * (9.0 / 5.0) + 32.0

            # Write the temperature to the file
            f_file.write(f"{f_temp}\n")

            # Print the conversion to the screen
            print(f"{temperature} Celcius equals {f_temp} Fahrenheit")

# Let the user know there was an exception
except Exception as e:
    print(f"A file exception occured. {e}")
```

Example run:

```
32.0 Celcius equals 89.6 Fahrenheit
0.0 Celcius equals 32.0 Fahrenheit
24.0 Celcius equals 75.2 Fahrenheit
-40 Celcius equals -40.0 Fahrenheit
```



JSON

Python has a built-in package called **json**, which can be used to work with JSON data.

```
import json
```

JSON (**J**ava**S**cript **O**bject **N**otation) is a popular data format used for representing structured data. It is a common format used to transmit and receive data over the internet or a local network between a server and application. It is a lightweight data interchange format that is easy for both humans and machines to read and write. In Python, the json module provides functions to work with JSON data.

- **Serialization and Deserialization:** The json module facilitates the conversion between Python data structures (such as lists and dictionaries) and a JSON-formatted string or a file.
    - **Serialization:**
        - **json.dumps()** converts a Python object into a JSON string.
        - **json.dump()** writes a Python object to a JSON file.
    - **Deserialization:**
        - **json_loads()** converts a JSON string into a Python object
        - **json_load()** reads a JSON into a Python object.
- **Compatible Data Types:** JSON supports several basic data types such as strings, numbers, booleans, arrays (lists), objects (dictionaries), and null values. Python's equivalent data structures (lists, dictionaries, etc.) can be serialized into JSON and vice versa.

Example of serializing (encoding) a Python object into JSON format:

```
import json

data = {"name": "John", "age": 30, "city": "New York"}
json_string = json.dumps(data)
print(json_string)
# Output: {"name": "John", "age": 30, "city": "New York"}
```

Example of deserializing (decoding) JSON into a Python object:

```
import json

json_string = '{"name": "John", "age": 30, "city": "New York"}'
data = json.loads(json_string)
print(data)
# Output: {'name': 'John', 'age': 30, 'city': 'New York'}
```

JSON in Python serves as a convenient and widely used format for data exchange between different systems, making it a valuable tool for communication and data storage in various applications.

Tutorial 10.8: Dumping and Loading a JSON File

Dump JSON to File

With the json module, there is a **dump** function that dumps data to a json file. We are going to write the Python list to a file. Any Python data structure can be used, such as a tuple, dictionary, list or a combination.

Create a file named **json_dump.py**

```python
# Simple json.dump() demo
# json can store data structures

# Import the 'json' module to work with JSON data
import json

# Define a list of numbers
numbers = [2, 3, 4, 6, 7, 89]

# Assign a filename to store the JSON data
FILENAME = "numbers.json"

try:
    # Open the file named 'numbers.json' in write mode ('w')
    # The 'with' statement ensures the file is properly closed after its use
    with open(FILENAME, "w") as file_handle:
        # Serialize the 'numbers' list into JSON format
        # and write it into the file
        json.dump(numbers, file_handle)

    # Let the user know there was an exception
except Exception as e:
    print(f"A file exception occured. {e}")
```

- **import json** - allows the use of JSON-related functions and methods.
- **numbers** - is a Python list containing some numerical values.
- **FILENAME** - stores the name of the file where the JSON data will be saved.
- **with open(FILENAME, 'w') as file_handle:** - opens the file 'numbers.json' in write mode. The 'with' statement ensures proper handling and closure of the file after its use.
- **json.dump(numbers, file_handle)** - serializes the numbers list into JSON format and writes it into the file 'numbers.json'. This stores the list content in JSON format within the file.

Load JSON Data

The Python json load() method loads our list back into memory and displays the data structure.

```python
# Simple json.load() demo
# json can load data structures
# Import the 'json' module to work with JSON data
import json

# Assign the filename 'numbers.json' to the variable FILENAME
FILENAME = "numbers.json"

try:
    # Open the file 'numbers.json' in read mode ('r')
    # The 'with' statement ensures the file is properly closed after its use
    with open(FILENAME) as file_handle:
        # Load the JSON data from the file into the variable 'numbers'
        numbers = json.load(file_handle)

    # Print the content stored in the 'numbers' variable,
    # which contains the JSON data
    print(numbers)

# Let the user know there was an exception
except Exception as e:
    print(f"A file exception occured. {e}")
```

Example run:

```
[2, 3, 4, 6, 7, 89]
```

- **import json** - allows the use of functions and methods from the 'json' module in Python.
- **FILENAME = "numbers.json"** - assigns the name of the file containing JSON data to the variable FILENAME.
- **with open(FILENAME) as file_name:** - opens the file 'numbers.json' in read mode. The 'with' statement ensures proper handling and closure of the file after its use.
- **json.load(file_handle)** - loads the JSON data from the file 'numbers.json' into the variable numbers.
- **print(numbers)** - displays the content of the numbers variable, which holds the JSON data loaded from the file.

Text Files and Binary Files

Python can create files for humans to read, and it can also create files that are intended only for computers to read.

In the previous steps you have been writing and reading data to text files. In each project, text files were created and saved in such a manner that a human can read them if they want to. In reality the text files have already been interpreted by the computer. Computers do not understand human language, and so values are written in a way that a computer can process, known as binary.

Binary files can be used to store any data; for example, a JPEG image is a binary file designed to be read by a computer system. The data inside a binary file is stored as raw bytes and is not readable by humans.

Here is a short program. Read the code line by line; can you work out what the code will do?

```
data = [100, 24, 255]
buffer = bytes(data)
print(buffer)
f = open("binary.txt", "bw")
f.write(buffer)
f.close()
```

How does this code work?

The program starts by creating a list called data containing three numbers. The data list is then converted to bytes (binary numbers) and stored in the buffer variable, which is then printed to the screen.

A new file called binary.txt is created and the mode set to **bw** so that bytes can be written into the file, making it a binary file.

The contents of the buffer variable are written to the file and finally the file is closed.

Copy the code above into your Python editor, save it, and run the code. You can look at the contents of binary.txt using a text editor.

Your text editor tried to open the file but has failed to read the contents correctly. This is because the text editor is trying to interpret the file contents as text.

Reading a Binary File

Reading binary files is very similar to reading text files. Have a look at this code and see if you can work out what it does.

```
f = open("binary.txt", "br")
binary = f.read()
print(binary)
data = list(binary)
print(data)
f.close()
```

This code opens the file and sets the mode to binary read, "br". It then reads the contents of the file into the variable binary and prints it to the screen.

The data type of the binary variable is bytes. The bytes data is converted into a list using the list() function and stored in the variable data.

When you run this code, you should see the following:

- The contents of the file, printed to the screen as bytes
- The original list, converted back from the binary data

Assignment 1: Serialize Yourself Baby

1. Create a Python dictionary with some personal information (name, age, hobbies).
2. Serialize this dictionary to a JSON file using **json.dump**
3. Load the data from the file using **json.load** and print it.
4. Use try except to catch and display any exceptions.

Example run:

```
Data has been serialized to personal_info.json
Loaded data from personal_info.json:
{'name': 'William Loring', 'age': 69, 'hobbies': ['reading', 'robots', 'coding']}
```

Glossary

catch To prevent an exception from terminating a program using the try and except statements.

newline A special character used in files and strings to indicate the end of a line.

Pythonic A technique that works elegantly in Python. "Using try and except is the Pythonic way to recover from missing files".

Quality Assurance A person or team focused on ensuring the overall quality of a software product. QA is often involved in testing a product and identifying problems before the product is released.

text file A sequence of characters stored in permanent storage like a hard drive.

Assignment Submission

1. Attach the pseudocode or create a TODO.
2. Attach all tutorials and assignments.
3. Attach screenshots showing the successful operation of each tutorial program.
4. Submit in Blackboard.