File handling (hour 9)

- File handling in Python.
- Reading from and writing to files.
- Working with text and CSV files.

Common File Formats

Common file formats encompass a wide range of data storage formats that are commonly used across various industries and applications. Here are some of the most prevalent file formats:

- **Text Files (.txt):** Simple files containing plain text without specific formatting. They are widely used for storing notes, configurations, logs, and other textual information.
- **CSV Files (.csv):** Comma-Separated Values files store tabular data, with each row representing a record and values separated by commas or other delimiters. They are often used for data exchange between different applications and for data analysis.
- **JSON Files (.json):** JavaScript Object Notation files store data in a hierarchical structure using key-value pairs. They are commonly used for configuration files, web APIs, and data exchange between web applications.
- **Binary Files:** These files store data in binary format, making them more efficient for certain types of data, such as images, videos, audio, and compiled program files.
- **Image Files:** Various formats like JPEG (.jpg), PNG (.png), GIF (.gif), and BMP (.bmp) are used to store images with different levels of compression and color depth.

Common File Formats

- **Video Files:** Formats like MP4 (.mp4), AVI (.avi), and MOV (.mov) are used for storing video content with various compression techniques.
- **Audio Files:** Formats like MP3 (.mp3), WAV (.wav), and FLAC (.flac) are used for storing audio data with different levels of compression and audio quality.
- **Database Files:** Formats like SQLite (.sqlite), MySQL (.mysql), and PostgreSQL (.pg) are used for storing structured data in databases.
- **Microsoft Office Documents:** Formats like DOCX (.docx) for Word documents, XLSX (.xlsx) for Excel spreadsheets, and PPTX (.pptx) for PowerPoint presentations are widely used in office environments.
- **PDF Files (.pdf):** Portable Document Format files are used for documents that need to be preserved in a consistent layout across different devices and platforms.
- **HTML Files (.html, .htm):** Hypertext Markup Language files are used to create web pages and are viewed in web browsers.
- **XML Files (.xml):** Extensible Markup Language files store data in a hierarchical structure using custom tags. XML is commonly used for data exchange and configuration in various applications.

Opening Files

File handling in Python is a crucial aspect of working with data and information. It allows you to read, write, and manipulate files on your computer. Python provides a built-in `open()` function to work with files. This function allows you to specify the file name, the mode of access, and other optional parameters.

To open a file, you use the `open()` function, which returns a file object. The basic syntax for opening a file is as follows

```
file = open(filename, mode)
```

Here, `filename` is the name of the file you want to open, and `mode` is the access mode in which you want to open the file.

File Modes in Python

The most commonly used ones are:

- **Read mode ('r')**: This mode allows you to read the contents of a file. If the file does not exist, it will raise a `FileNotFoundError`.
- **Write mode ('w')**: This mode allows you to write data to a file. If the file already exists, it will truncate (empty) the file. If the file doesn't exist, a new file will be created.
- **Append mode ('a')**: This mode allows you to append new data to the end of an existing file. If the file doesn't exist, a new file will be created.
- **Text mode ('t')**: This mode is used in conjunction with read, write, or append modes to work with text data (default mode).
- **Binary mode ('b')**: This mode is used in conjunction with read, write, or append modes to work with binary data (e.g., images, audio files).

Combine modes when opening a file

```
file = open(filename, 'mode1mode2')

file = open('example.txt', 'wb')
file = open('example.txt', 'rt')
```

- **Read and Write** ('r+'): Opens the file for both reading and writing. The file pointer is placed at the beginning of the file.
- **Write and Read** ('w+'): Opens the file for both writing and reading. If the file exists, it is truncated to zero length. If it does not exist, a new file is created.
- **Append and Read** ('a+'): Opens the file for both appending and reading. The file pointer is placed at the end of the file.

Closing Files

To close the file after you're done working with it, you should use the `close()` method of the file object:

```
file.close()
```

It is essential to close the file once you're done working with it to free up system resources.

Writing to a File and Closing It

Let's demonstrate how to open a file in write mode, write some content, and then close it:

```
text = """some test multiline text
line 1: blablabla
line 2: la-la-la"""“

file = None
try:
    file = open('example.txt', 'w')
    file.write(text)
except Exception as e:
    print(f"An error occurred: {e}")
finally:
    if file is not None:
        file.close()
```

Reading from a File and Closing It

Let's demonstrate how to open a file in read mode, read its contents, and then close it:

```
try:
    file = open('example.txt', 'r')
    content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found!")
finally:
    file.close()
```

with statements

It's good practice to use with statements, which automatically closes the file for you after leaving the block:

```
text = """“
some test multiline text
line 1: blablabla
line 2: la-la-la
"""“
try:
    with open('example.txt', 'w') as file:
        file.write(text)
except Exception as e:
    print(f"An error occurred: {e}")
```

with statements

```
try:
 with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found!")
```

Reading from Files

- Using `read()` to read the entire file: Suppose we have a file named example.txt with the following content:

Line 1: This is the first line.
Line 2: This is the second line.
Line 3: This is the third line

```
# Using read() to read the entire file content
try:
    with open('example.txt', 'r') as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("File not found!").
```

Reading from Files

- Using `readline()` to read lines one by one:

```
# Using readline() to read lines one by one
try:
    with open('example.txt', 'r') as file:
        line = file.readline()
        while line:
            # strip() removes the newline characters
            print(line.strip())
            line = file.readline()
except FileNotFoundError:
    print("File not found!")
```

Reading from Files

- Using `readlines()` to read all lines at once into a list:

```
# Using readlines() to read all lines at once into a list
try:
    with open('example.txt', 'r') as file:
        lines = file.readlines()
        for line in lines:
            print(line.strip())
except FileNotFoundError:
    print("File not found!")
```

Writing to Files

- `write()` method: This method is used to write a single string to the file.

```
# Using write() to write a single line
data_to_write = "This is a single line.\n“

try:
    with open('new_file.txt', 'w') as file:
        file.write(data_to_write)
except Exception as e:
    print(f"An error occurred: {e}")
```

Writing to Files

- `writeline()` method: There is no built-in method called `writeline()` in Python. The correct method to use for writing a single line is `write()`.
- `writelines()` method: This method is used to write a list of strings to the file. Each string in the list will be written as a separate line in the file.

```
# Using writelines() to write multiple lines
lines_to_write = [
    "This is the first line.\n",
    "This is the second line.\n",
    "This is the third line.\n"]
try:
    with open('new_file.txt', 'w') as file:
        file.writelines(lines_to_write)
except Exception as e:
    print(f"An error occurred: {e}")
```

seek() method

In Python, the `seek()` method is used to move the file pointer to a specific position within a file. The file pointer represents the current position in the file where the next read or write operation will take place. The `seek()` method allows you to reposition the file pointer to any desired location, allowing you to read or write data from or to a specific location in the file.

The general syntax of the `seek()` method is as follows:

```
file.seek(offset, whence)
```

- **offset**: The number of bytes to move the file pointer. The offset can be positive or negative. A positive offset moves the file pointer forward, and a negative offset moves it backward.
- **whence** (optional): The reference point from where the offset is applied. It can take one of the following values:

  0 (default): The offset is relative to the beginning of the file.

  1: The offset is relative to the current file position.

  2: The offset is relative to the end of the file.

Example

```
file = open('example.txt', 'w+')

# Writing to the file
file.write("This is a new line.\n")

# Moving the file cursor to the beginning (since writing moved it to the end)
file.seek(0)

# Reading the content from the file
content = file.read()
print(content)  # Output: "This is a new line.\n"file.close()
```

Moving the File Pointer to the Beginning

```
with open('example.txt', 'r') as file:
    # Move the file pointer to the beginning of the file
    file.seek(0)

    # Read and print the first 10 characters of the file
    content = file.read(10)
    print(content)
```

Moving the File Pointer Relative to Current Position

```
with open('example.txt', 'r') as file:
    # Read and print the first 5 characters of the file
    content = file.read(5)    print(content)  # Output: "This “
    # Move the file pointer 3 characters forward from the current position
    file.seek(3, 1)
    # Read and print the next 7 characters
    content = file.read(7)
    print(content)  # Output: "is a ne"
```

Moving the File Pointer Relative to the End of File

```
with open('example.txt', 'r') as file:
    # Move the file pointer 5 characters backward from the end of the file
    file.seek(-5, 2)
    # Read and print the last 5 characters of the file
    content = file.read()
    print(content)  # Output: "ine.\n“
```

Content of 'example.txt':

This is a new line.

File handling with *print* and *input* functions

In Python, you can handle file input and output using the built-in functions `print()` and `input()`.

While these functions are commonly used for standard input and output, we can also use them to read from and write to files.

Writing to a File using *print()*

To write data to a file using `print()`, we need to specify the file object as the value for the file parameter. The `print()` function will then write the data to the file instead of printing it to the console.

```
# Writing data to a file using print()
data_to_write = "This is some data that will be written to the file.“

try:
    with open('output.txt', 'w') as file:
        print(data_to_write, file=file)
except Exception as e:
    print(f"An error occurred: {e}")
```

Writing to a File using *print()*

To print to a file, we can redirect the standard output stream to a file using the `sys.stdout` module in Python. This way, any output that would normally be printed to the console will be written to the specified file.

```
import sys
data_to_write = "This is some data that will be printed to the file.“
try:
    with open('output.txt', 'w') as file:
        sys.stdout = file
        print(data_to_write)
except Exception as e:
    print(f"An error occurred: {e}")
finally:
    # Resetting the standard output back to the console
    sys.stdout = sys.__stdout__
```

Reading from a File using *input()*

We can read data from a file using the `input()` function by redirecting the standard input stream to the file. This can be achieved using the `sys.stdin` module.

```
import sys
# Reading data from a file using input()
try:
    with open('input.txt', 'r') as file:
        sys.stdin = file
        data = input()
        print("Data read from the file:", data)
except FileNotFoundError:
    print("File not found!")
finally:
    # Resetting the standard input back to the console
    sys.stdin = sys.__stdin__
```

CSV (Comma-Separated Values) *Files*

CSV files are a specific type of text file used to store tabular data, where each line represents a row, and the values within each row are separated by a delimiter, typically a comma (`,`), semicolon (`;`), or tab (`\t`). CSV files are widely used for data exchange between different systems and applications, especially in data analysis and database management. Python provides a csv module to easily work with CSV files, making it simple to read and write data in a structured manner.

Reading Data from a CSV *File*

Reading Text Files Line by Line and Processing Data.

Suppose we have a CSV file named data.csv with the following content:

```
Name,Age,Occupation
John,30,Engineer
Alice,25,Teacher
Bob,28,Doctor
```

Reading Data from a CSV File

```
try:
    with open('data.csv', 'r') as file:
        lines = file.readlines()
        # Extracting header and removing newline characters
        header = lines[0].strip().split(',')
        data = []
        # Extracting data rows and splitting by commas
        for line in lines[1:]:
            row = line.strip().split(',')
            data.append(row)
        # Printing the data in a structured manner
        for row in data:
            name, age, occupation = row
            print(f"Name: {name}, Age: {age}, Occupation: {occupation}")
except FileNotFoundError:
    print("File not found!")
```

Writing Data to a CSV File

```
data_to_write = [
    ['Name', 'Age', 'Occupation'],
    ['John', '30', 'Engineer'],
    ['Alice', '25', 'Teacher'],
    ['Bob', '28', 'Doctor']]
try:
    with open('new_data.csv', 'w') as file:
        for row in data_to_write:
            # Joining the elements of the row with commas
            # and adding a newline character
            line = ','.join(row) + '\n‘
            file.write(line)
except Exception as e:
    print(f"An error occurred: {e}")
```

Reading and Writing Data to CSV Files using the csv Module

```
import csv

try:
    with open('data.csv', 'r') as csv_file:
        csv_reader = csv.reader(csv_file)
        header = next(csv_reader)  # Read the header row
        for row in csv_reader:
            name, age, occupation = row
            print(f"Name: {name}, Age: {age}, Occupation:
{occupation}")
except FileNotFoundError:
    print("File not found!")
```

Writing Data to a CSV File using the csv Module

```
import csv
data_to_write = [
    ['Name', 'Age', 'Occupation'],
    ['John', 30, 'Engineer'],
    ['Alice', 25, 'Teacher'],
    ['Bob', 28, 'Doctor']]
try:
    with open('new_data.csv', 'w', newline='') as csv_file:
        csv_writer = csv.writer(csv_file)
        for row in data_to_write:
            csv_writer.writerow(row)
except Exception as e:
    print(f"An error occurred: {e}")
```