Lecture 09 - Input and Output (I/O) Operations

1. Overview

Input and output operations (I/O) are essential in programming.

They allow programs to

- **Interact** live with users
- Take data as **input**
- Display or save the results as **output**

This notebook covers

- How to take input from users
- How to display output using the `print()` function
- File I/O operations: reading from and writing to files
- Handling CSV files with basic file operations
- Working with CSV/Excel files using `pandas`

2. Live inputs

2.1 The `input()` function

The `input()` function allows the user to provide input to the program while running by returning the input as a `string`.

In [ ]:
```
# Example: Asking for user input
name = input("What is your name? ")
print("Hello, " + name + "!")
```

2.2 Casting inputs

Results from `input()` can be casted from `string` to other data types (`int`, `float`, etc.).

In [ ]:
```
# Example: Taking a number as input
age = int(input("How old are you? "))  # Convert input to an integer
print("You are", age, "years old.")
print ("You were born in ", 2025 - age)
```

In [ ]:
```
# Example: Taking a float as input
balance = float(input("Enter your account balance: "))
print("Your balance is $", balance)
```

2.3 Multiple inputs

When multiple inputs are entered, they are returned in one line. They can be then separated into separate items using `split()`.

In [ ]:
```
# Example: Taking multiple inputs
first_name, last_name = input("Enter your first and last name: ").split
print("Hello, " + first_name + " " + last_name)
```

3. Live output

3.1 The `print()` function

The `print()` function displays outputs in the prompt terminal.

In [ ]:
```
# Example: Basic output using print()
print("Hello, world!")
```

3.2 Printing variables and expressions

The `print()` handles variables and expressions.

In [ ]:
```
# Example: Printing variables
name = "Alice"
age = 25
print("Name:", name, "Age:", age)
```

String formatting

`Python` provides several ways to format outputs as a combination of strings and variables.

- **Using commas:**

```
  print("string", variable, "string",
etc.)
```

In [ ]:
```
# Example: Using commas
print("Your balance is", balance, "dollars.")
```

- **Using f-strings (available in Python 3.6+):**

```
print (f"string {variable}")
```

In [ ]:
```
# Example: Using f-strings
print(f"Hello, {name}. You are {age} years old.")
```

- **Using the `format()` method:**

```
print ("string {}, {}".format(variable1,
variable2))
```

In [ ]:
```
# Example: Using format()
print("Hello, {}. You are {} years old.".format(name, age))
```

4. File input and output (File I/O)

Programs can **read from** files and **write to** files.

4.1 Opening files

Before reading from or writing to a file, a file object must be created. The `open()` function opens files and creates access in `Python`.

```
open (name, mode)
```

- The name (or path) of the file
- The access mode (e.g., `'r'` for reading, `'w'` for writing, `'a'` for appending)

Note: opened files must then be closed with `.close()` to clear the access.

In [ ]:
```
file_path = "Data/09/"
file_name = "example.txt"
file = open(file_path + file_name, "r")  # Open the file in read mode
```

4.2 Reading from

Several methods exist for reading file contents, such as `.read()`, `.readline()`, and `.readlines()`.

- **`read()`**: Reads the entire file.

In [ ]:
```
content = file.read()  # Read the entire file
print(content)
file.close()  # Close the file after reading
```

In [ ]:
```
content
```

- **`readline()`**: Reads one line at a time.

In [ ]:
```
file = open(file_path + file_name, "r")
line = file.readline()
while line:
    print(line.strip())  # Remove newline characters
    line = file.readline()
file.close()
```

- **`readlines()`** : Reads all lines and returns them as a list.

In [ ]:
```
lines
```

In [ ]:
```
file = open(file_path + file_name, "r")
lines = file.readlines()
for line in lines:
    print(line.strip())
file.close()
```

4.3 `with` for file handling

To avoid misusing file access (opening and closing), the `with` statement handles files more efficiently by automatically closing the file after the block of code is executed.

```
with expression as variable:
        # Code block that uses the resource (e.g., a file or
connection)
```

In [ ]:

```
# Example: Using 'with' to handle files
with open(file_path + file_name, "r") as file:
    content = file.read()
    print(content)
# No need to explicitly close the file; it is done automatically
```

4.4 Writing to

Writing data to a file is done using the `write()` or `writelines()` methods.

- The `'w'` (write) mode overwrites the existing content.
- The `'a'` (append) mode writes at the end of the file.

- **Write mode `m`:**

In [ ]:
```
# Example: Writing to a file
file_name = "new_file.txt"
with open(file_path + file_name, "w") as file: # Open file in write moc
    file.write("This is the first line.\n")
    file.write("This is the second line.\n")
```

In [ ]:
```
# checking
with open(file_path + file_name, "r") as file: # Open file in write moc
    print(file.read())
```

- **Append mode `a`:**

In [ ]:
```
# Example: Writing to a file
file_name = "new_file.txt"
with open(file_path + file_name, "a") as file: # Open file in write moc
    file.write("This is the first line.\n")
    file.write("This is the second line.\n")
    file.write("This is an additional line.\n")
```

In [ ]:
```
# checking
with open(file_path + file_name, "r") as file: # Open file in write moc
    print(file.read())
```

5. `.csv` files

`CSV` (Comma-Separated Values) files are the standard storing format for **tabular data**.

- The firt line contains the name of columns.
- Each line is a row where cells are separated by `,`.

Example: `"csv_example.csv"`

```
Name,Age,Balance
Alice,30,1000.50
Bob,25,1500.75
Charlie,35,2000.25
Diana,28,1800.60
Eve,22,1200.00
```

5.1 with `Python`

Handling `.csv` files from `Python` can be done using the `csv` module.

In [ ]:
```
import csv
```

Reading from

Reading a `csv` file is done using the `csv.reader()` method.

In [ ]:
```
import csv
# Example: Reading a CSV file
file_path = "Data/09/"
csv_file_name = 'csv_example.csv'
with open(file_path + csv_file_name, 'r') as file:
    csv_reader = csv.reader(file)
    for row in csv_reader:
        print(row)
```

`csv` files can also be directly read as `dict` objects using the `csv.DictRead()` method.

In [ ]:

```
# Example: Reading a CSV file with headers
with open(file_path + csv_file_name, 'r') as file:
    csv_reader = csv.DictReader(file)
    for row in csv_reader:
        print(f"{row['Name']} is {row['Age']} years old.")
```

Writing to

Writing data to a `csv` file is done by

1. Creating a `csv.writer()` object with the output file.
2. Inserting rows with `.writerow()`

- `w` mode

In [ ]:

```
# Example: Writing to a CSV file
csv_output_file_name = 'csv_output_example.csv'
with open(file_path + csv_output_file_name, 'w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerow(['Name', 'Age', 'Balance'])
    csv_writer.writerow(['Alice', '30', '1000.50'])
    csv_writer.writerow(['Bob', '25', '1500.75'])
```

- `a` mode

In [ ]:
```
csv_output_file_name = 'csv_output_example.csv'
with open(file_path + csv_output_file_name, 'a', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerow(['Name', 'Age', 'Balance'])
    csv_writer.writerow(['Charlie', '35', '2000.25'])
    csv_writer.writerow(['Eve', '22', '1200.00'])
```

In [ ]:
```
with open(file_path + csv_output_file_name, 'r') as file:
    csv_reader = csv.DictReader(file)
    for row in csv_reader:
        print(f"{row['Name']} is {row['Age']} years old.")
```

5.2 with `pandas`

`Pandas` simplifies reading, writing, and analyzing data, especially with `csv` files.

In [ ]:
```
import pandas as pd
```

Reading from

Reading a `csv` file is done using the `pd.read_csv()` method.

```
pd.read_csv(...)
```

Check method documentation (huge!).

Key parameters:

- `filepath_or_buffer` : The path to the file you want to read.
- `delimiter` or `sep` : The character that separates values (default is a comma).
- `header` : Row number to use as the column names.
- `names` : A list of column names to use.
- `index_col` : Column(s) to set as index.
- `skiprows` : Number of rows to skip at the start of the file.

In [ ]:

```
csv_file_name = 'csv_example.csv'
df = pd.read_csv(file_path + csv_file_name)
df
```

Example with Parameters

In [ ]:

```
# Customizing CSV reading
df = pd.read_csv(file_path + csv_file_name, delimiter=",", header=0, ir
df.head()
```

Writing to

Saving a `DataFrame` to a `csv` file is done using the `pd.to_csv()` method.

In [ ]:
```
# Example: Writing DataFrame to a CSV file
df_file_name = 'df_output_example.csv'
df.to_csv(file_path + df_file_name, index=False)  # `index=False` to ex
```

6. Excel files

Excel files are commonly used for data storage and transfer.

`Pandas` makes it easy to write and read these files.

6.1 Writing to

`Pandas` writes `DataFrames` to Excel files using the `to_excel()` method.

```
pd.to_excel(...)
```

Check method documentation (huge!).

Key parameters:

- `excel_writer`: This is the path to the Excel file ("output.xlsx") or an `ExcelWriter` object if saving multiple sheets to a single file.
- `sheet_name`: The name of the sheet where the `DataFrame` will be saved. Default is "Sheet1".
- `na_rep`: Defines how missing values (`NaN`) should appear in the Excel file (e.g., na_rep="N/A").
- `columns`: List of column names to write; writes all columns by default.
- `header`: If `True`, includes column headers; set to `False` to exclude.
- `index`: If `True`, includes the `DataFrame`'s index; if `False`, omits it.
- `startrow` and `startcol`: Define the starting cell (row and column) for writing data.

Basic example of writing to Excel

Note: The `to_excel()` method requires the `openpyxl` library for `.xlsx` files (install with `!pip install openpyxl` if necessary).

In [ ]:

```
import pandas as pd
import openpyxl

# Sample DataFrame
data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "Balance": [1000.50, 1500.75, 1200.30]
}
df = pd.DataFrame(data)

# Writing the DataFrame to an Excel file
file_path = "Data/09/"
excel_output_name = "excel_output_example.xlsx"
df.to_excel(file_path + excel_output_name, index=False)  # `index=False
```

Writing multiple sheets to an Excel File

The `ExcelWriter` class allows to save multiple DataFrames to different sheets within the same Excel file.

In [ ]:

```
# Create additional sample DataFrames
data2 = {
    "Product": ["Widget A", "Widget B", "Widget C"],
    "Price": [20.5, 45.3, 30.0]
}
df2 = pd.DataFrame(data2)

with pd.ExcelWriter(file_path + excel_output_name) as writer:
    df.to_excel(writer, sheet_name="Customer Data", index=False)
    df2.to_excel(writer, sheet_name="Product Data", index=False)
```

Specifying columns and formatting

Customizing which columns to write and add formatting options like `headers` are also possible.

In [ ]:
```
# Writing selected columns only
excel_output_2_name = "excel_output_selected_columns.xlsx"
df.to_excel(file_path + excel_output_2_name, columns=["Name", "Balance"
```

6.2 Reading from an Excel file

Reading a `xlsx` file is done using the `pd.read_excel()` method.

```
pd.read_excel()
```

Check method documentation (huge!)

- `sheet_name` : Name or index of the sheet to read.
- `header` , `names` , `index_col` : Similar to read_csv().
- `usecols` : Specifies columns to load.

In [ ]:

```
# Reading an Excel file
df = pd.read_excel(file_path + excel_output_name, sheet_name="Customer
df.head()
```

Example: Reading a specific sheet with column selection

In [ ]:

```
# Reading specific columns from an Excel sheet
df = pd.read_excel(file_path + excel_output_name, sheet_name="Product D
df.head()
```

🚦 Checkpoint

Reading, Manipulating, and Saving Data with `pandas`

Goal: In this exercise, you'll read the CSV file, manipulate the data using `pandas`, and save the modified data back to a CSV file.

```
file_name = "checkpoint_stock_data_large.csv"
```

- Import data and inspect in `DataFrame`
- **Task 1:** Filter out all rows where the stock price is greater than USD 2000.
- **Task 2:** Add a new column called `Value` which is calculated as `Price * Volume`.
- **Task 3:** Group the data by the `Stock` column and calculate the average price for each stock.
- Save the manipulated DataFrame to a new CSV file.