**Madiha Aimon Tappal**

[**madihaaimon@gmail.com**](mailto:madihaaimon@gmail.com)

**Data Engineering Batch – 1**

**Day – 8 Assignment**

**Python**

**CSV Files: -**

CSV files are widely used for data storage and exchange due to their simplicity and ease of use. Each line in a CSV file represents a record, and the values within a record are separated by a delimiter (commonly a comma). CSV files can be easily created and manipulated using spreadsheet software or programming languages.

**Reading CSV Files with Python: -**

Python provides several ways to read CSV files, and one of the most popular libraries for this purpose is csv. However, when dealing with more complex data structures and operations, the pandas library is often preferred.

**Using csv Module: -**

The csv module is a built-in Python module that provides functionality to both read from and write to CSV files.

**Example: -**

import csv

# Reading a CSV file

with open('example.csv', 'r') as file:

reader = csv.reader(file)

for row in reader:

print(row)

#### **Using pandas: -**

The pandas library simplifies the process of reading CSV files and working with tabular data.

import pandas as pd

# Reading a CSV file into a DataFrame

df = pd.read\_csv('example.csv')

print(df)

### Reading CSV Files: -

#### **Using** csv **Module:**

The **csv** module in Python provides a **reader** object that can be used to read data from a CSV file.

### Example:

import csv

# Reading a CSV file

with open('example.csv', 'r') as file:

reader = csv.reader(file)

for row in reader:

print(row)

In this example:

* The **open ()** function is used to open the CSV file in read mode (**'r'**).
* The **csv. reader** is then used to create a reader object, and you can iterate over the rows in the CSV file.

#### **Using pandas:**

The **pandas** library simplifies reading CSV files by directly creating a DataFrame from the data.

**Example:**

import pandas as pd

# Reading a CSV file into a DataFrame

df = pd.read\_csv('example.csv')

print(df)

In this example:

* The **pd. read\_csv ()** function reads the CSV file and creates a DataFrame.
* The resulting DataFrame is printed, displaying the tabular data.

### Writing CSV Files:

#### **Using csv Module:**

The **csv** module also provides a **writer** object that can be used to write data to a CSV file.

Example:

import csv

# Writing to a CSV file

data = [['Name', 'Age', 'City'],

['John', 25, 'New York'],

['Jane', 30, 'San Francisco'],

['Bob', 22, 'Los Angeles']]

with open('output.csv', 'w', newline='') as file:

writer = csv.writer(file)

writer.writerows(data)

In this example:

* The **open()** function is used to open a new CSV file in write mode (**'w'**).
* The **csv.writer** is then used to create a writer object, and **writerows()** writes the data to the file.

#### **Using** pandas**:**

The **pandas** library makes writing to a CSV file straightforward.

**Example:**

import pandas as pd

# Creating a DataFrame

data = {'Name': ['John', 'Jane', 'Bob'],

'Age': [25, 30, 22],

'City': ['New York', 'San Francisco', 'Los Angeles']}

df = pd.DataFrame(data)

# Writing DataFrame to a CSV file

df.to\_csv('output.csv', index=False)

In this example:

* A DataFrame is created using a dictionary.
* The **to\_csv()** method of the DataFrame is used to write the DataFrame to a CSV file, and **index=False** ensures that the row indices are not written.

#### csv **Module:**

* The **delimiter** parameter in **csv.reader** and **csv.writer** can be used to specify a custom delimiter (e.g., **delimiter='\t'** for tab-separated values).

#### pandas**:**

* The **pd.read\_csv()** function in **pandas** supports various parameters such as **sep** (to specify the delimiter), **header** (to specify the row to use as column names), and many more.
* The **to\_csv()** method in **pandas** provides options like **header** (to include or exclude column names in the output), **index** (to include or exclude row indices), and **sep** (to specify the delimiter).

### Use Cases of CSV Files:

1. **Data Exchange:** CSV is a common format for exchanging data between different systems. It's widely supported by spreadsheet applications, databases, and various data processing tools.
2. **Data Storage:** CSV files are simple and lightweight, making them suitable for storing structured data. They can be easily edited with a text editor and can serve as a human-readable data format.
3. **Data Analysis:** Analysts and data scientists often use CSV files as an intermediate step for loading data into tools like Pandas for analysis and manipulation.
4. **Web Development:** CSV files are frequently used for data exchange between web applications and databases. They are easy to generate and parse, making them useful for web development tasks.
5. **Configuration Files:** CSV files can be used to store configuration data for applications. This is common when the configuration involves structured data with multiple parameters.

### File IO using Python:

#### **Reading from a File:**

with open('example.txt', 'r') as file:

content = file.read()

print(content)

* **Modes in open () Function:**
  + **'r'**: Read (default mode) - Opens the file for reading.
  + **'w'**: Write - Opens the file for writing. Creates a new file or truncates an existing file.
  + **'a'**: Append - Opens the file for writing. Creates a new file or appends to an existing file.
  + **'b'**: Binary mode - Used for binary files (e.g., images).
* **with Statement:**
  + Ensures proper file handling by automatically closing the file when the code block is exited.

#### **Writing to a File:**

# Open a file in write mode ('w')

with open('output.txt', 'w') as file:

file.write('Hello, this is a sample text.')

In this example:

* The open() function is used to open a file named 'output.txt' in write mode ('w').
* The with statement ensures proper file handling.
* The write() method is used to write the specified text to the file.

### *Appending to a File:*

# Open a file in append mode ('a')

with open('output.txt', 'a') as file:

file.write('\nAppending more text to the file.')

In this example:

* The open() function is used to open the same file ('output.txt') in append mode ('a').
* The with statement ensures proper file handling.
* The write() method appends the specified text to the end of the file.

### Reading Lines from a File:

# Open a file in read mode ('r')

with open('example.txt', 'r') as file:

lines = file.readlines()

for line in lines:

print(line.strip()) # strip() removes newline characters

In this example:

* The **readlines()** method reads all the lines of the file and returns them as a list.
* The **for** loop iterates over each line, and **strip()** is used to remove newline characters.

**Using with Statement:**

The **with** statement is used for proper handling of resources, such as file handling. It automatically takes care of closing the file when the block of code is exited.

with open('example.txt', 'r') as file:

content = file.read()

# Code block for working with the file

# File is automatically closed outside the with block

Using the **with** statement is considered good practice, as it avoids potential issues related to not closing files explicitly.

**Processing Python Lists: -**

Processing Python lists involves performing various operations on the elements of a list, such as iterating, filtering, mapping, and reducing. Let's explore some common operations:

### Iterating Over a List:

You can use a **for** loop to iterate over the elements of a list:

numbers = [1, 2, 3, 4, 5]

for num in numbers:

print(num)

### List Comprehension:

List comprehension is a concise way to create lists. It combines the **for** loop and the creation of a new list in a single line:

numbers = [1, 2, 3, 4, 5]

doubled\_numbers = [x \* 2 for x in numbers]

print(doubled\_numbers)

### Filtering a List:

You can use the **filter()** function to create a new list containing only the elements that satisfy a given condition. Often, a lambda function is used with **filter()** for concise filtering:

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even\_numbers)

### Mapping Elements of a List:

The **map()** function is used to apply a given function to all the items in a list:

numbers = [1, 2, 3, 4, 5]

squared\_numbers = list(map(lambda x: x\*\*2, numbers))

print(squared\_numbers)

### Reducing a List:

The **reduce()** function from the **functools** module is used to successively apply a function to the elements of a list, reducing it to a single value:

from functools import reduce

numbers = [1, 2, 3, 4, 5]

sum\_of\_numbers = reduce(lambda x, y: x + y, numbers)

print(sum\_of\_numbers)

These are fundamental operations when working with lists in Python.

### Lambda Functions in Python:

Lambda functions, also known as anonymous functions, are defined using the **lambda** keyword. They are used for small, one-time operations without the need for a full function definition.

Example:

# Regular function

def add(x, y):

return x + y

# Equivalent lambda function

add\_lambda = lambda x, y: x + y

print(add(5, 3)) # Output: 8

print(add\_lambda(5, 3)) # Output: 8

Lambda functions are concise and are often used in situations where a short function is needed temporarily.

* Lambda Syntax:
  + lambda parameters: expression
* Comparison with Regular Function:
  + Lambda functions are anonymous, one-liners used for short-term operations.

### Usage of Lambda Functions:

Lambda functions are commonly used with higher-order functions like **map()**, **filter()**, and **reduce()**.

**Higher-Order Functions:**

* Functions that take other functions as arguments or return functions.
* Lambda functions are commonly used with higher-order functions.

### Filter Data in Python Lists using filter() and Lambda:

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Using filter() and lambda

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even\_numbers)

* **filter() Function:**
  + Filters elements based on the specified function (in this case, the lambda function).
* **Lambda Function in filter():**
  + The lambda function is applied to each element in the list, keeping only those that satisfy the condition.

**Use of Lambda Function in Python:**

* **Temporary Functions:**
  + Lambda functions are suitable for short, temporary operations where a full function definition is unnecessary.

**Practical Uses of Python Lambda Function:**

* **Sorting:**
  + Lambda functions are often used as key functions in sorting operations.

students = [('John', 25), ('Jane', 30), ('Bob', 22)]

sorted\_students = sorted(students, key=lambda x: x[1])

print(sorted\_students)

* **GUI Programming:**
  + Lambda functions are used in event handling for GUI applications.

button = Button(text="Click me", command=lambda: print("Button clicked"))

### Using lambda() Function with map(), filter(), reduce():

numbers = [1, 2, 3, 4, 5]

# Using map() with lambda to square each number

squared\_numbers = list(map(lambda x: x\*\*2, numbers))

# Using filter() with lambda to keep even numbers

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

# Using reduce() with lambda to find the sum

from functools import reduce

sum\_of\_numbers = reduce(lambda x, y: x + y, numbers)

print(squared\_numbers)

print(even\_numbers)

### print(sum\_of\_numbers)

* **map():**
  + Applies the lambda function to each element in the list.
* **filter():**
  + Filters elements based on the lambda function.
* **reduce():**
  + Reduces the list to a single value by applying the lambda function cumulatively.

Lambda functions in Python are anonymous functions created using the **lambda** keyword. They are typically used for short-term operations where a full function definition is not required. Lambda functions are especially useful when you need a quick function for a specific task.

### 1. ****As Arguments to Higher-Order Functions:****

Lambda functions are often used as arguments to higher-order functions like **map()**, **filter()**, and **sorted()**.

#### **Example with** map()**:**

numbers = [1, 2, 3, 4, 5]

squared\_numbers = list(map(lambda x: x\*\*2, numbers))

print(squared\_numbers)

This squares each element in the list.

### 2. ****Filtering Data:****

Lambda functions are commonly used with **filter()** to filter elements based on certain conditions.

#### **Example with** filter()**:**

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even\_numbers)

This filters out only the even numbers from the list.

### 3. ****Sorting with**** sorted()****:****

Lambda functions are useful when custom sorting criteria are needed.

#### **Example with** sorted()**:**

students = [('John', 25), ('Jane', 30), ('Bob', 22)]

sorted\_students = sorted(students, key=lambda x: x[1])

print(sorted\_students)

This sorts a list of students based on their age.

### 4. ****Anonymous Functions in GUI Programming:****

Lambda functions can be used in GUI programming, especially for event handling.

#### **Example with Tkinter:**

from tkinter import Tk, Button

root = Tk()

button = Button(root, text="Click me", command=lambda: print("Button clicked"))

button.pack()

Here, a lambda function is used as the command when the button is clicked.

### 5. ****Conditional Expressions:****

Lambda functions can be used for concise conditional expressions.

get\_result = lambda x: "Even" if x % 2 == 0 else "Odd"

print(get\_result(4)) # Output: Even

print(get\_result(7)) # Output: Odd

### 6. ****Mathematical Operations:****

Lambda functions are handy for quick mathematical operations.

add = lambda x, y: x + y

multiply = lambda x, y: x \* y

print(add(3, 5)) # Output: 8

print(multiply(2, 4)) # Output: 8

### 7. ****Functional Programming:****

In functional programming paradigms, lambda functions are used extensively.

# Using map() and lambda to find squares

numbers = [1, 2, 3, 4, 5]

squares = list(map(lambda x: x\*\*2, numbers))

print(squares)

Lambda functions contribute to creating more functional and concise code.

**Advantages of Lambda Functions in Python:**

1. **Conciseness:**

* Lambda functions are concise and can be defined in a single line, making the code more compact.

1. **Readability in Higher-Order Functions:**

* Lambda functions work well with higher-order functions like **map()**, **filter()**, and **sorted()**, improving readability.

1. **Avoiding Function Name Conflicts:**

* Lambda functions are anonymous, eliminating the need to define a function name. This helps avoid potential naming conflicts.

1. **Functional Programming Paradigms:**

* Lambda functions align well with the principles of functional programming, allowing for more expressive and functional code.