**Python Introduction**

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

**1. Write programs to parse text files, CSV, HTML, XML and JSON documents and extract relevant data. After retrieving data check any anomalies in the data, missing values etc.**

**Parsing CSV Files and Checking for Missing Values:**

import csv

with open("covid\_19\_clean\_complete.csv", 'r') as csvfile:

data=[]

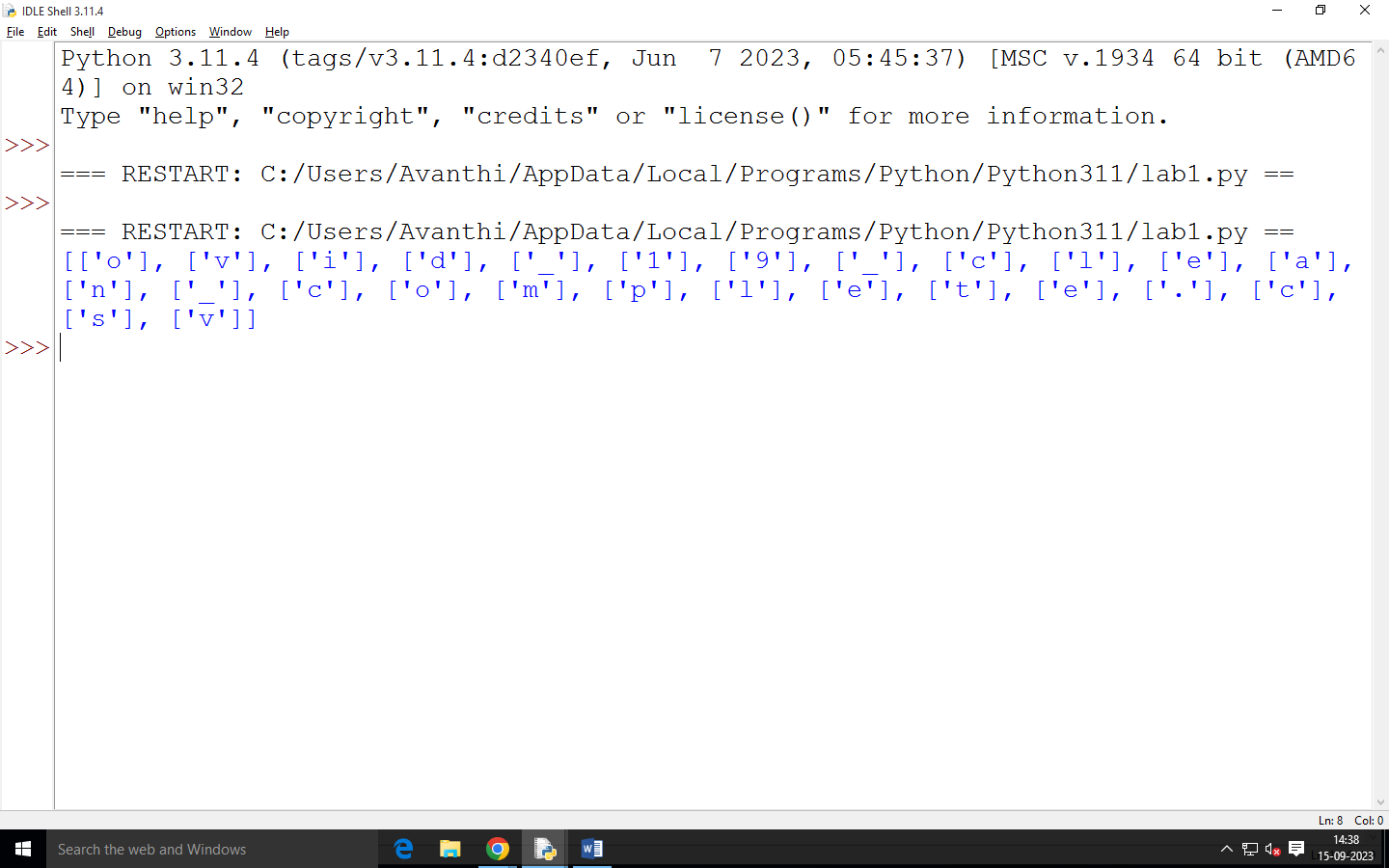
reader = csv.reader("covid\_19\_clean\_complete.csv")

header = next(reader) # Read the header

for row in reader:

data.append(row)

print(data)



def check\_missing\_values(data):

missing\_values = []

for row in data:

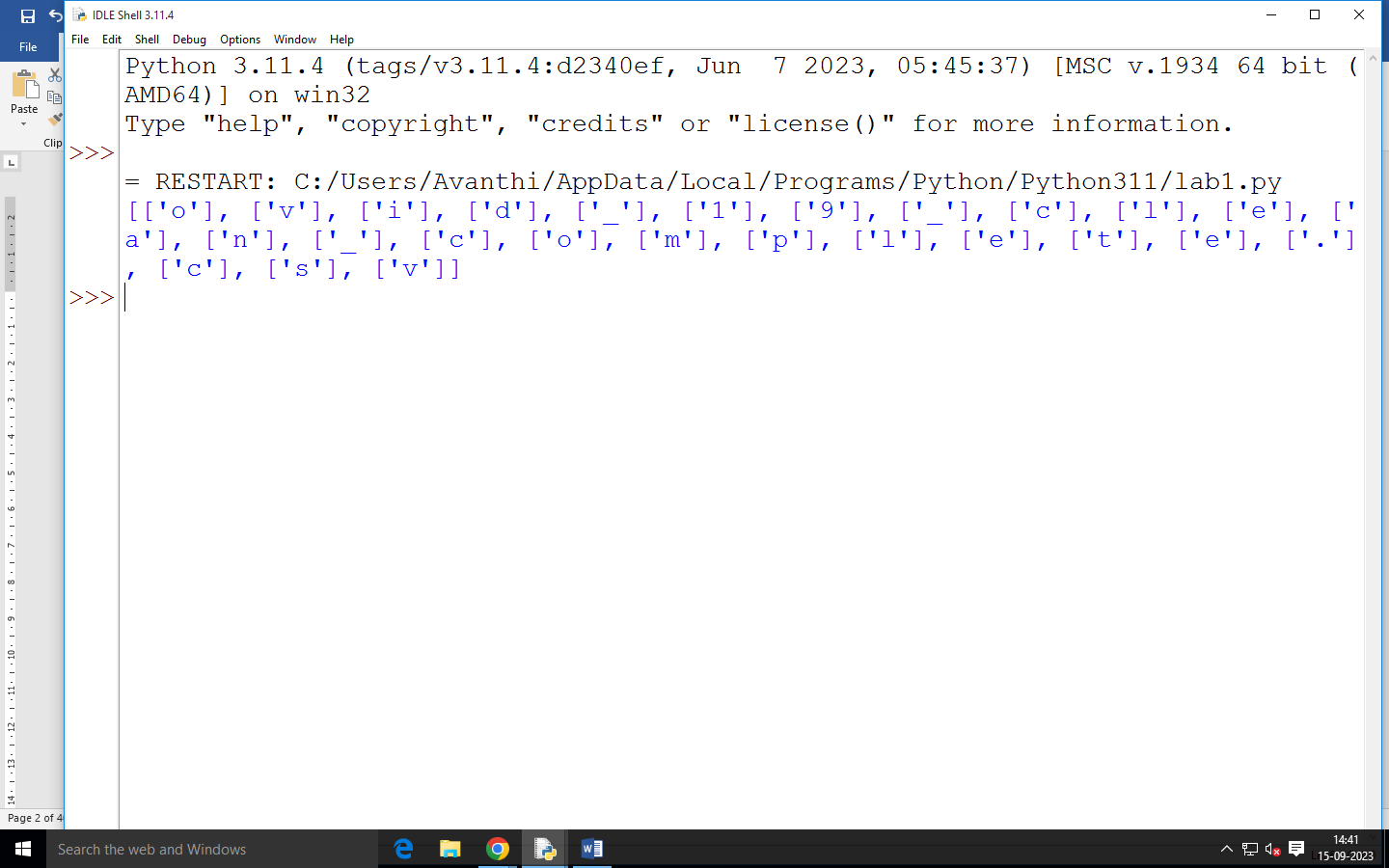
for value in row:

if not value:

missing\_values.append(row)

break

return missing\_values



**Parsing HTML Files Using Beautiful Soup:**

You'll need to install the beautifulsoup4 library. You can install it using pip install beautifulsoup4.

# import module

import requests

import pandas as pd

from bs4 import BeautifulSoup

# link for extract html data

def getdata(url):

r = requests.get(url)

return r.text

htmldata = getdata("https://www.geeksforgeeks.org/how-to-automate-an-excel-sheet-in-python/?ref=feed")

soup = BeautifulSoup(htmldata, 'html.parser')

data = ' '

for data in soup.find\_all("p"):

print(data.get\_text())

Parsing XML Files Using ElementTree:

import xml.etree.ElementTree as ET

def parse\_xml\_file(file\_path):

tree = ET.parse(file\_path)

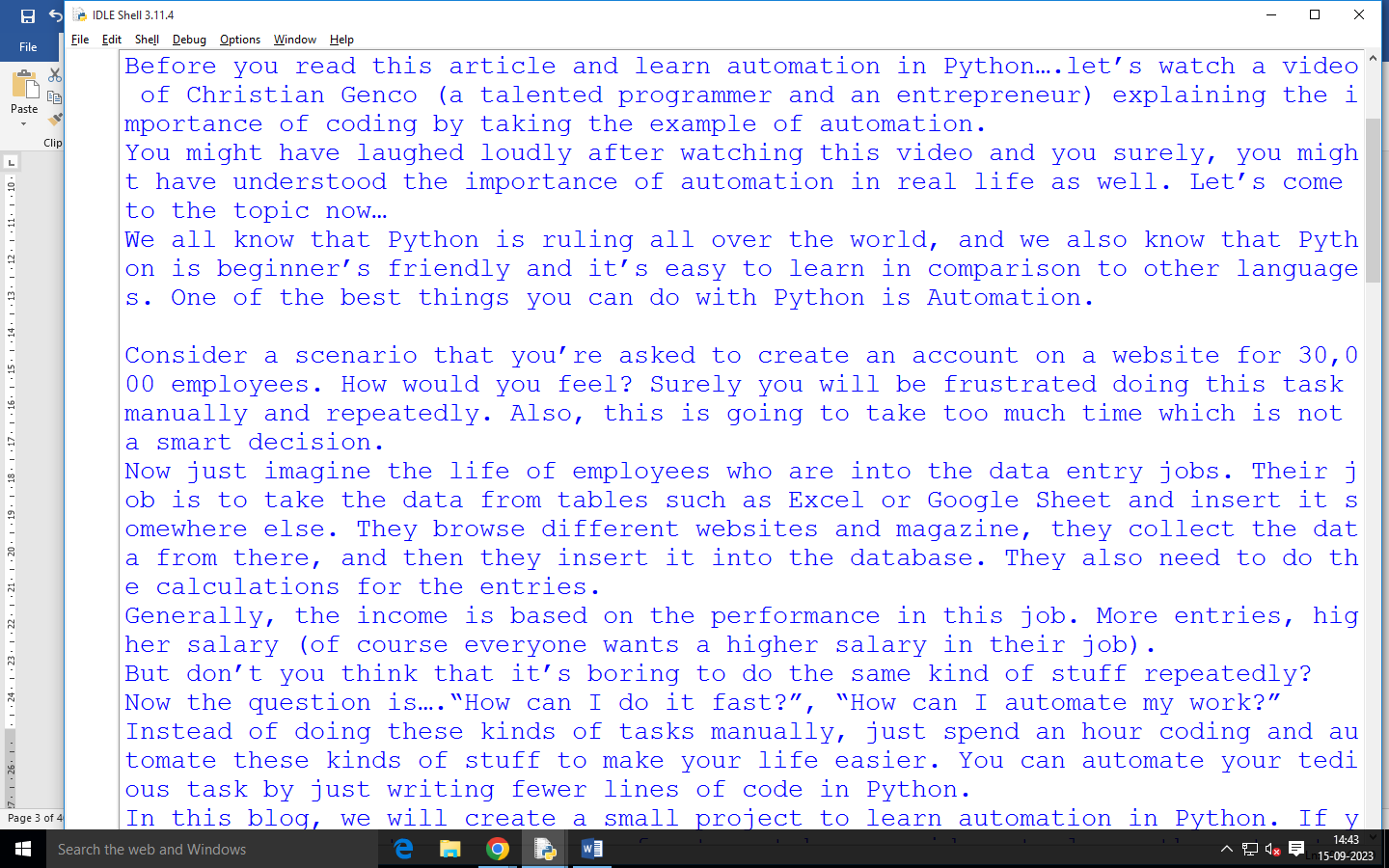
root = tree.getroot()

return root

xml\_file\_path = 'data.xml'

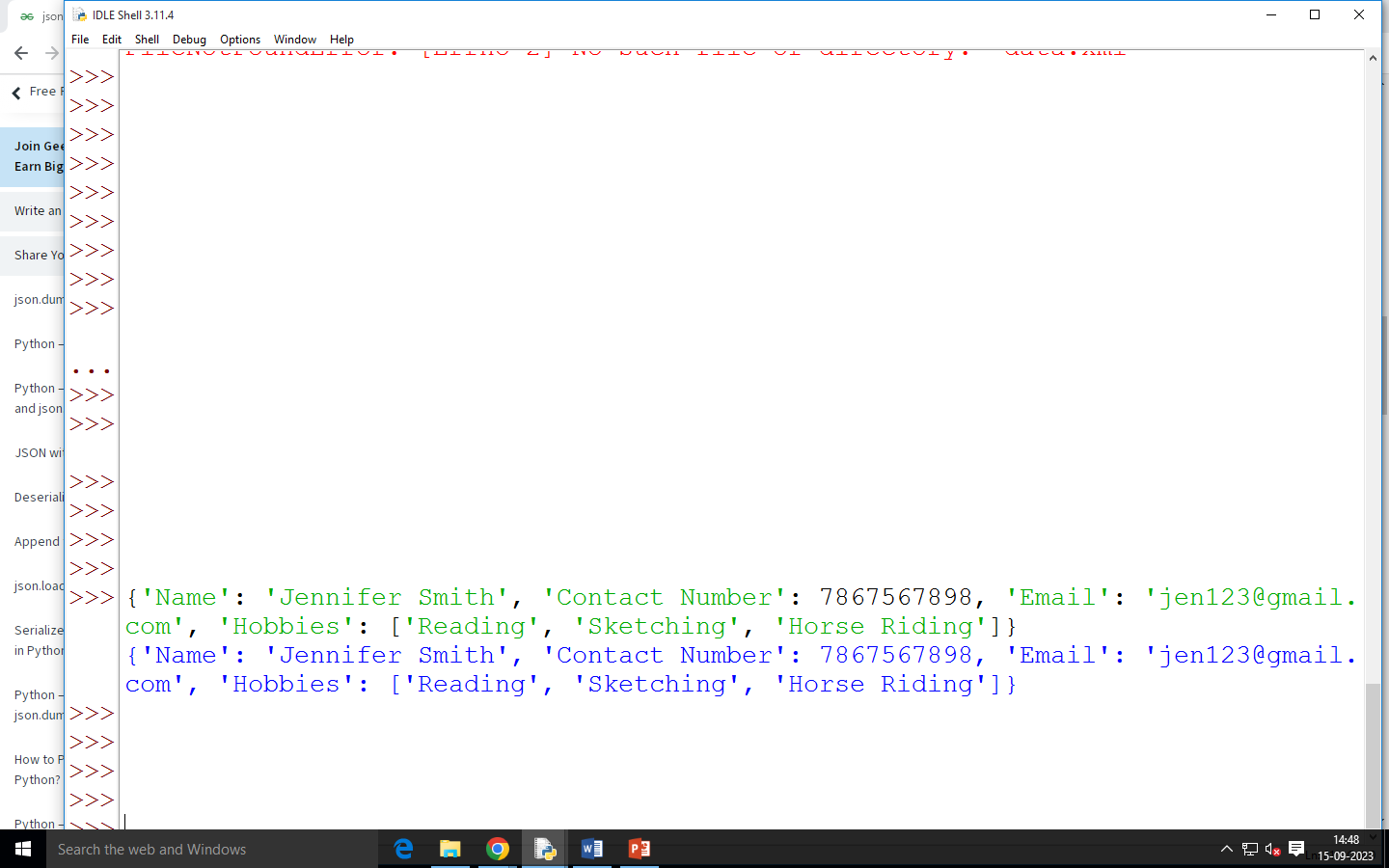
parsed\_xml = parse\_xml\_file(xml\_file\_path)

print(ET.tostring(parsed\_xml, encoding='utf-8').decode('utf-8')) # Print parsed XML content



**Parsing JSON Files and Checking for Anomalies:**

* # Python program to read
* # json file
* import json
* # Opening JSON file
* f = open('data.json')
* # returns JSON object as
* data = json.load(f)
* # Iterating through the json
* # list
* for i in data['emp\_details']:
* print(i)
* # Closing file
* f.close()



## Write Bytes to File in Python

**some\_bytes = b'\xC3\xA9'**

**# Open in "wb" mode to**

**# write a new file, or**

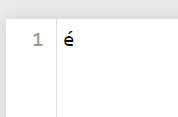
**# "ab" mode to append**

**with open("my\_file.txt", "wb") as binary\_file:**

**# Write bytes to file**

**binary\_file.write(some\_bytes)**

**output:**



**Write programs for searching, splitting, and replacing strings based on pattern matching using regular expressions**

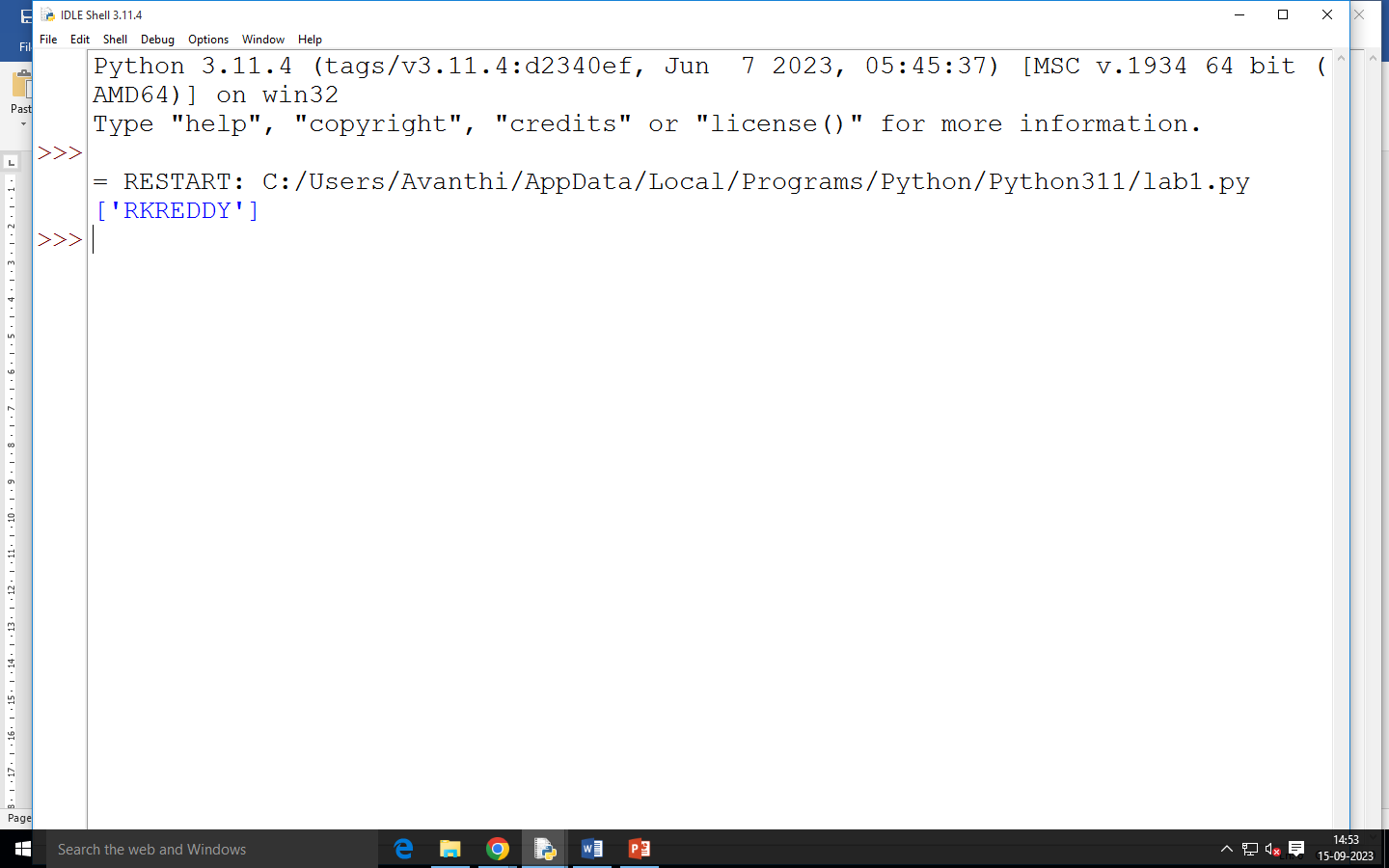
import re

xx = "RKREDDY is master in python"

r1 = re.findall(r"^\w+",xx)

print(r1)

output



import re

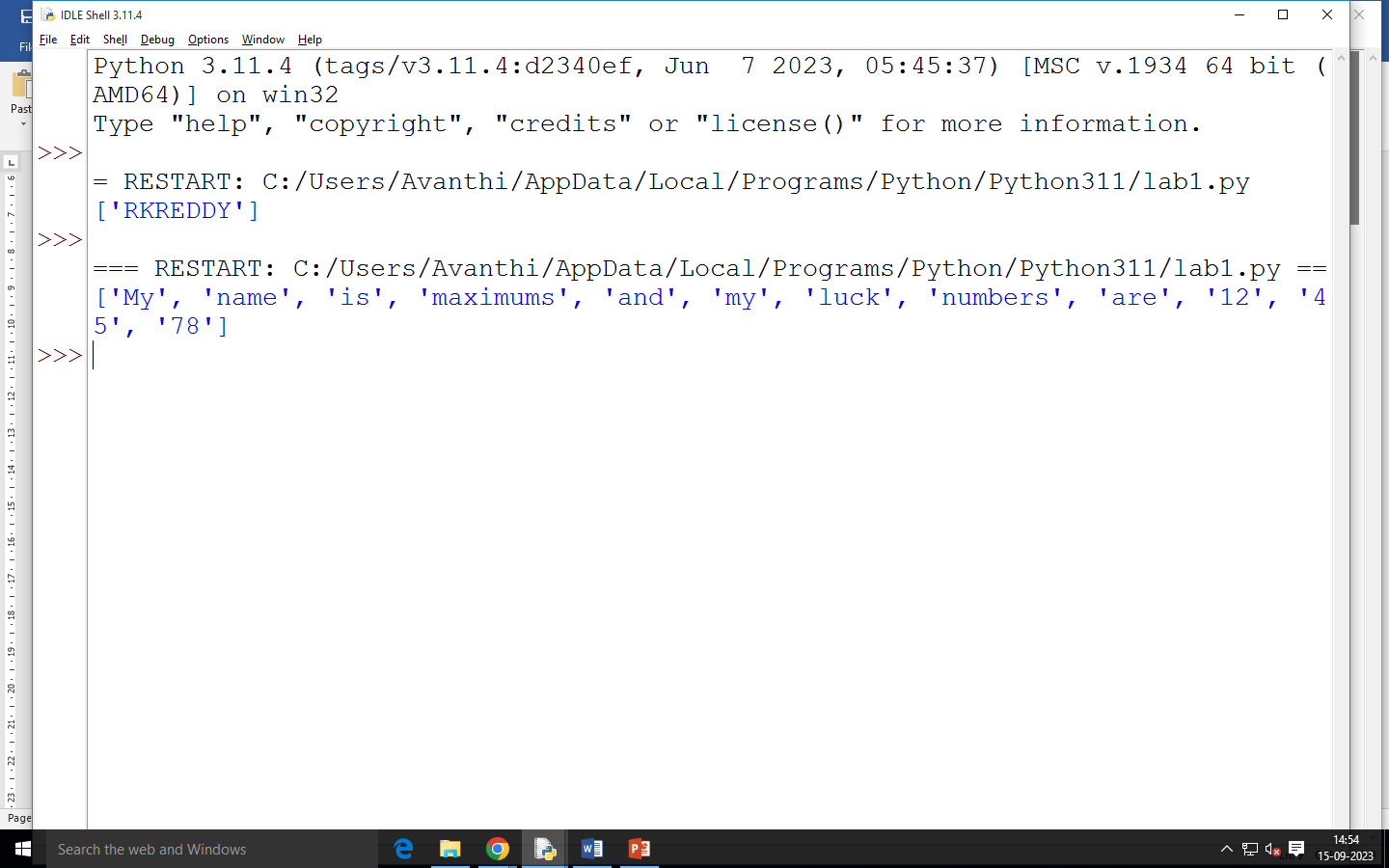
target\_string = "My name is maximums and my luck numbers are 12 45 78"

# split on white-space

word\_list = re.split(r"\s+", target\_string)

print(word\_list)

output:



import re

target\_str = "Jessa knows testing and machine learning"

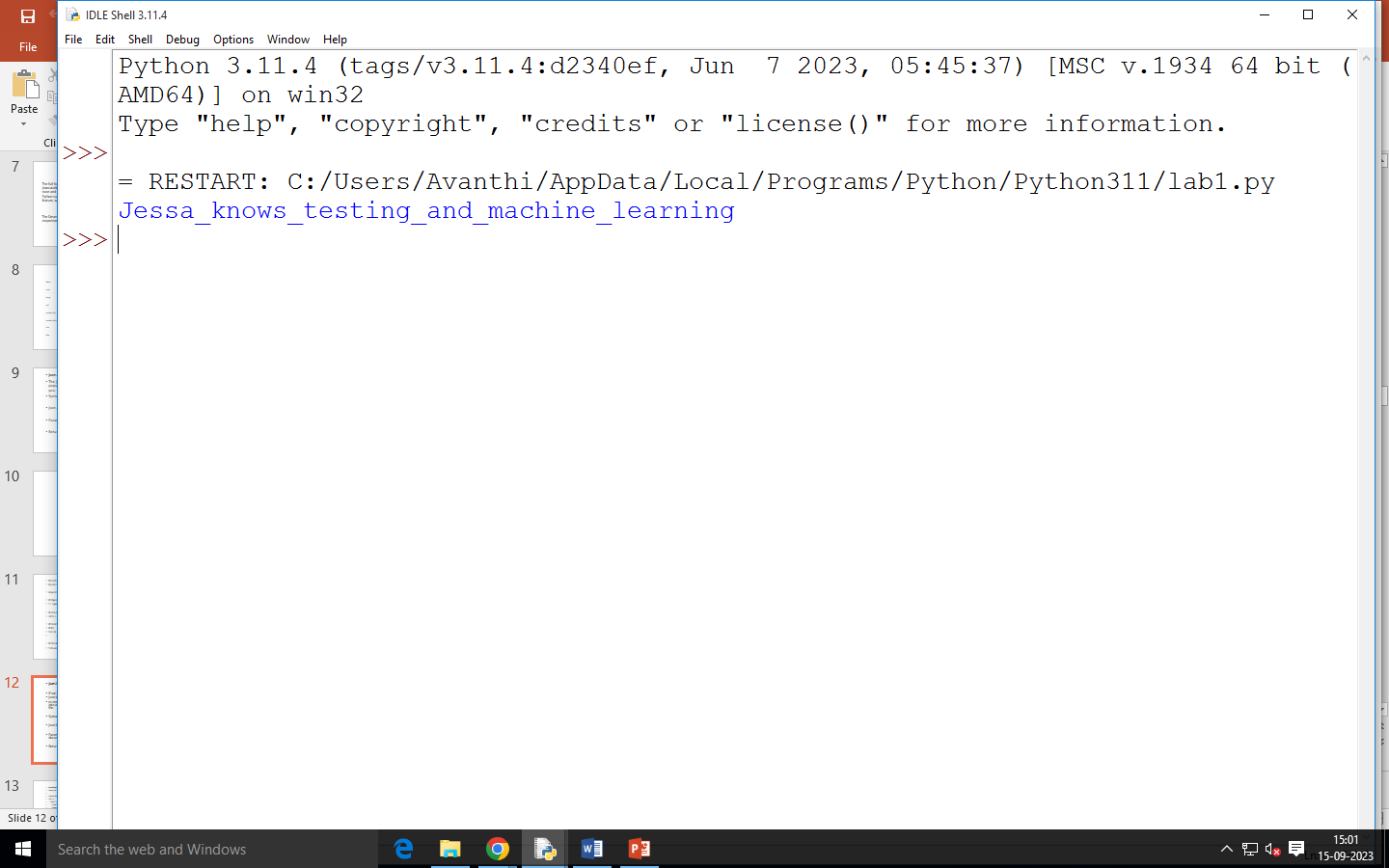
res\_str = re.sub(r"\s", "\_", target\_str)

# String after replacement

print(res\_str)

# Output 'Jessa\_knows\_testing\_and\_machine\_learning'

Output:



**Design a relational database for a small application and populate the database. Using SQL do the CRUD (create, read, update and delete) operations**

**CRUD:**

design a simple relational database schema and perform CRUD operations using SQL. Let's assume we're creating a small application to manage a library's book collection. The database will need tables to store information about books and authors. Here's how the schema could look:

**Tables:**

**Authors**

author\_id (Primary Key)

first\_name

last\_name

**Books**

book\_id (Primary Key)

title

publication\_year

author\_id (Foreign Key referencing Authors)

**Creating the Database:**

CREATE DATABASE LibraryApp;

USE LibraryApp;

CREATE TABLE Authors (

author\_id INT PRIMARY KEY,

first\_name VARCHAR (50),

last\_name VARCHAR (50)

);

CREATE TABLE Books (

book\_id INT PRIMARY KEY,

title VARCHAR(100),

publication\_year INT,

author\_id INT,

FOREIGN KEY (author\_id) REFERENCES Authors(author\_id)

);

**Populating the Database:**

INSERT INTO Authors (author\_id, first\_name, last\_name)

VALUES

(1, 'John', 'Doe'),

(2, 'Jane', 'Smith');

INSERT INTO Books (book\_id, title, publication\_year, author\_id)

VALUES

(1, 'The Book Title', 2020, 1),

(2, 'Another Book', 2018, 2);

**CRUD Operations:**

**1. Create (INSERT):**

INSERT INTO Authors (author\_id, first\_name, last\_name)

VALUES (3, 'Michael', 'Johnson');

INSERT INTO Books (book\_id, title, publication\_year, author\_id)

VALUES (3, 'New Book', 2022, 3);

**2. Read (SELECT):**

-- Get all books with their authors

SELECT Books.title, Authors.first\_name, Authors.last\_name

FROM Books

JOIN Authors ON Books.author\_id = Authors.author\_id;

-- Get books published after 2019

SELECT title, publication\_year

FROM Books

WHERE publication\_year > 2019;

**3. Update (UPDATE):**

-- Update book title

UPDATE Books

SET title = 'Updated Title'

WHERE book\_id = 2;

-- Update author's last name

UPDATE Authors

SET last\_name = 'Johnson'

WHERE author\_id = 1;

**4. Delete (DELETE):**

-- Delete a book

DELETE FROM Books

WHERE book\_id = 3;

**-- Delete an author and their books**

DELETE FROM Authors

WHERE author\_id = 3;

**Create a Python MongoDB client using the Python module pymongo. Using a collection object practice functions for inserting, searching, removing, updating, replacing, and aggregating documents, as well as for creating indexes**

create a Python MongoDB client using the pymongo module and practice various functions on a collection:

Reasons to opt for MongoDB :

1. It supports hierarchical data structure (Please refer docs for details)

2. It supports associate arrays like Dictionaries in Python.

3. Built-in Python drivers to connect python-application with Database. Example- PyMongo

4. It is designed for Big Data.

5. Deployment of MongoDB is very easy.

**Create a connection :** The very first after importing the module is to create a MongoClient.

from pymongo import MongoClient

client = MongoClient(“mongodb://localhost:27017/”)

EXAMPLE

from pymongo import MongoClient

# Connect to MongoDB

client = MongoClient('mongodb://localhost:27017/')

db = client['mydatabase'] # Replace 'mydatabase' with your database name

collection = db['mycollection'] # Replace 'mycollection' with your collection name

# Insert documents

data\_to\_insert = [

{"name": "Alice", "age": 30, "city": "New York"},

{"name": "Bob", "age": 25, "city": "San Francisco"},

{"name": "Charlie", "age": 40, "city": "Los Angeles"}

]

collection.insert\_many(data\_to\_insert)

# Find documents

result = collection.find({"city": "New York"})

for doc in result:

print(doc)

# Update documents

collection.update\_one({"name": "Alice"}, {"$set": {"age": 31}})

# Replace document

collection.replace\_one({"name": "Bob"}, {"name": "Bobby", "age": 26, "city": "San Francisco"})

# Remove documents

collection.delete\_one({"name": "Charlie"})

# Aggregation

pipeline = [

{"$group": {"\_id": "$city", "count": {"$sum": 1}}}

]

aggregation\_result = collection.aggregate(pipeline)

for doc in aggregation\_result:

print(doc)

# Create an index

collection.create\_index("name")

# Close the connection

client.close()

**Write programs to create numpy arrays of different shapes and from different sources, reshape and slice arrays, add array indexes, and apply arithmetic, logic, and aggregation functions to some or all array elements**

**What is NumPy?**

NumPy is a Python library used for working with arrays.

It also has functions for working in domain of linear algebra, fourier transform, and matrices.

NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

NumPy stands for Numerical Python.

**Why Use NumPy?**

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

import numpy as np

# Create arrays of different shapes and sources

array1d = np.array([1, 2, 3, 4, 5]) # 1D array

array2d = np.array([[1, 2, 3], [4, 5, 6]]) # 2D array

array\_zeros = np.zeros((3, 4)) # 3x4 array of zeros

array\_ones = np.ones((2, 2)) # 2x2 array of ones

array\_range = np.arange(0, 10, 2) # Array with values [0, 2, 4, 6, 8]

# Reshape arrays

reshaped\_array = array2d.reshape(3, 2) # Reshape 2x3 array to 3x2

# Slicing arrays

sliced\_array = array1d[1:4] # Slice elements from index 1 to 3 (inclusive)

sliced\_2d\_array = array2d[:, 1] # Slice second column from 2D array

# Add array indexes

indexed\_array = array1d + np.arange(5) # Add array indexes to each element

# Arithmetic operations

addition\_result = array1d + 10

multiplication\_result = array2d \* 2

# Logic operations

boolean\_array = array1d > 3 # Array of boolean values based on condition

# Aggregation functions

sum\_array = np.sum(array1d) # Sum of all elements

mean\_array = np.mean(array2d) # Mean of all elements

max\_value = np.max(array1d) # Maximum value in the array

print("1D Array:", array1d)

print("2D Array:", array2d)

print("Reshaped Array:", reshaped\_array)

print("Sliced Array:", sliced\_array)

print("Sliced 2D Array:", sliced\_2d\_array)

print("Indexed Array:", indexed\_array)

print("Addition Result:", addition\_result)

print("Multiplication Result:", multiplication\_result)

print("Boolean Array:", boolean\_array)

print("Sum of Array:", sum\_array)

print("Mean of Array:", mean\_array)

print("Maximum Value:", max\_value)

**Write programs to use the pandas data structures:** Frames and series as storage containers and for a variety of data-wrangling operations, such as:

**• Single-level and hierarchical indexing**

**Single-Level Indexing:**

Single-level indexing is the basic way of indexing in Pandas, where you have a single index to access rows and columns of a DataFrame or Series.

**DataFrame Single-Level Indexing:**

import pandas as pd

data = {'A': [1, 2, 3], 'B': [4, 5, 6]}

df = pd.DataFrame(data, index=['row1', 'row2', 'row3'])

# Accessing a single column using label-based indexing

column\_a = df['A'] # Returns a Series

# Accessing a single row using label-based indexing

row\_1 = df.loc['row1'] # Returns a Series

# Accessing a specific cell using label-based indexing

cell\_value = df.at['row2', 'B'] # Returns a scalar value

Series Single-Level Indexing:

import pandas as pd

data = [10, 20, 30]

s = pd.Series(data, index=['a', 'b', 'c'])

**# Accessing a single element using label-based indexing**

value = s['b'] # Returns a scalar value

**Hierarchical Indexing (MultiIndexing):**

Hierarchical indexing allows you to have multiple levels of indexing, which is especially useful when dealing with multi-dimensional or hierarchical data. It's used to index and manipulate data in higher-dimensional structures.

**DataFrame Hierarchical Indexing:**

import pandas as pd

data = {'A': [1, 2, 3], 'B': [4, 5, 6]}

index = pd.MultiIndex.from\_tuples([('row1', 'sub1'), ('row1', 'sub2'),

('row2', 'sub1'), ('row2', 'sub2')],

names=['index1', 'index2'])

df = pd.DataFrame(data, index=index)

# Accessing using MultiIndex

value = df.loc['row1', 'sub1'] # Returns a Series

Series Hierarchical Indexing:

import pandas as pd

data = [10, 20, 30, 40]

index = pd.MultiIndex.from\_tuples([('A', 'x'), ('A', 'y'),

('B', 'x'), ('B', 'y')],

names=['index1', 'index2'])

s = pd.Series(data, index=index)

# Accessing using MultiIndex

value = s.loc['A', 'y'] # Returns a scalar value

**Handling missing data**

Handling missing data is an important aspect of data analysis and manipulation in pandas, a popular Python library for data manipulation and analysis. Pandas provides several tools and techniques to handle missing data effectively. Here are some common approaches:

1. Detecting Missing Data: You can use various methods to detect missing data in a pandas DataFrame or Series:

import pandas as pd

df = pd.DataFrame({'A': [1, 2, None, 4], 'B': [5, None, 7, 8]})

# Check for missing values

print(df.isnull())

# Count missing values in each column

print(df.isnull().sum())

Dropping Missing Data: If the missing data is not critical and doesn't affect your analysis, you can drop rows or columns containing missing values using the dropna() method:

# Drop rows with any missing value

df\_dropped = df.dropna()

# Drop columns with any missing value

df\_dropped\_columns = df.dropna(axis=1)

Filling Missing Data: You can fill missing values with specific values using the fillna() method:

Filling null values with the previous ones

# importing numpy as np

import numpy as np

# dictionary of lists

dict = {'First Score':[100, 90, np.nan, 95],

'Second Score': [30, 45, 56, np.nan],

'Third Score':[np.nan, 40, 80, 98]}

# creating a dataframe from dictionary

df = pd.DataFrame(dict)

# filling missing value using fillna()

print(df.fillna(0))

(OR)

# importing pandas as pd

import pandas as pd

# importing numpy as np

import numpy as np

# dictionary of lists

dict = {'First Score':[100, 90, np.nan, 95],

'Second Score': [30, 45, 56, np.nan],

'Third Score':[np.nan, 40, 80, 98]}

# creating a dataframe from dictionary

df = pd.DataFrame(dict)

# filling a missing value with

# previous ones

print(df.fillna(method ='pad’))

#df.fillna(method ='bfill’)

Filling a null values using replace() method

# importing pandas package

import pandas as pd

# making data frame from csv file

data = pd.read\_csv("employees.csv")

# will replace Nan value in dataframe with value -99

data.replace(to\_replace = np.nan, value = -99)

Using interpolate() function to fill the missing values using linear method.

# importing pandas as pd

import pandas as pd

# Creating the dataframe

df = pd.DataFrame({"A":[12, 4, 5, None, 1],

"B":[None, 2, 54, 3, None],

"C":[20, 16, None, 3, 8],

"D":[14, 3, None, None, 6]})

# Print the dataframe

print(df)

**Arithmetic and Boolean operations on entire columns and tables**

you can perform arithmetic and Boolean operations on entire columns and tables using various built-in functions and operators. Pandas is a powerful library in Python for data manipulation and analysis. Here's how you can perform these operations:

1. Arithmetic Operations on Columns: You can perform arithmetic operations (addition, subtraction, multiplication, division, etc.) on entire columns in a Pandas DataFrame.

import pandas as pd

# Create a DataFrame

data = {'A': [1, 2, 3], 'B': [4, 5, 6]}

df = pd.DataFrame(data)

# Perform arithmetic operations on columns

df['C'] = df['A'] + df['B']

df['D'] = df['A'] \* 2

print(df)

**Boolean Operations on Columns:** You can perform Boolean operations (comparison, masking, etc.) on entire columns in a Pandas DataFrame.

import pandas as pd

# Create a DataFrame

data = {'A': [1, 2, 3], 'B': [4, 5, 6]}

df = pd.DataFrame(data)

# Boolean operations on columns

df['C'] = df['A'] > 1 # Creates a Boolean column based on a comparison

filtered\_df = df[df['B'] > 4] # Creates a new DataFrame with a Boolean mask

print(df)

print(filtered\_df)

**Applying Functions to Columns:** You can also apply custom functions to columns using the apply method.

import pandas as pd

# Create a DataFrame

data = {'A': [1, 2, 3], 'B': [4, 5, 6]}

df = pd.DataFrame(data)

# Applying functions to columns

def custom\_function(x):

return x \* 2

df['C'] = df['A'].apply(custom\_function)

print(df)

**Arithmetic Operations on Tables (DataFrames):** You can perform element-wise arithmetic operations on entire DataFrames.

import pandas as pd

# Create DataFrames

data1 = {'A': [1, 2, 3], 'B': [4, 5, 6]}

data2 = {'A': [2, 3, 4], 'B': [7, 8, 9]}

df1 = pd.DataFrame(data1)

df2 = pd.DataFrame(data2)

# Element-wise arithmetic operations on DataFrames

result\_df = df1 + df2

print(result\_df)

Database-type operations (such as merging and aggregation)

In pandas, a popular Python library for data manipulation and analysis, you can perform various database-type operations such as merging and aggregation on DataFrame and Series data structures. These operations are essential for working with structured data and conducting data analysis tasks. Here's an overview of how to perform merging and aggregation using pandas:

**Merging DataFrames:**

Merging is the process of combining two or more DataFrames based on common columns or indices. The primary function for merging DataFrames in pandas is the merge() function. Commonly used types of merges include:

1. Inner Merge: Retains only the rows with matching keys in both DataFrames.

2. Outer Merge: Includes all rows from both DataFrames, filling in missing values with NaN where necessary.

3. Left Merge: Keeps all rows from the left DataFrame and only matching rows from the right DataFrame.

4. Right Merge: Keeps all rows from the right DataFrame and only matching rows from the left DataFrame.

Example of performing an inner merge:

import pandas as pd

df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'value1': [1, 2, 3]})

df2 = pd.DataFrame({'key': ['B', 'C', 'D'], 'value2': [4, 5, 6]})

merged\_df = pd.merge(df1, df2, on='key', how='inner')

print(merged\_df)

Aggregation:

Aggregation involves summarizing data using functions like sum, mean, count, etc. Pandas provides the groupby() function to group data by one or more columns and then perform aggregation operations on the grouped data.

import pandas as pd

data = {'Category': ['A', 'B', 'A', 'B', 'A'],

'Value': [10, 20, 30, 40, 50]}

df = pd.DataFrame(data)

grouped = df.groupby('Category')

aggregated\_result = grouped['Value'].sum()

print(aggregated\_result)

You can also perform multiple aggregation functions at once:

aggregated\_result = grouped['Value'].agg(['sum', 'mean', 'count'])

print(aggregated\_result)

**Plotting individual columns and whole tables**

In pandas, you can easily plot individual columns and whole tables using the built-in plotting capabilities. Pandas provides a .plot() function that can be applied to both Series (individual columns) and DataFrames (whole tables). This function uses Matplotlib for visualization by default.

Here's how you can plot individual columns and whole tables using pandas:

python

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# Create a sample DataFrame

data = {

'A': np.random.rand(10),

'B': np.random.rand(10),

'C': np.random.rand(10)

}

df = pd.DataFrame(data)

# Plotting individual columns

df['A'].plot(kind='bar') # You can change the plot type as needed (e.g., 'line', 'scatter', etc.)

plt.title('Individual Column A')

plt.xlabel('Index')

plt.ylabel('Value')

plt.show()

# Plotting whole table

df.plot(kind='line') # This will plot all columns as lines

plt.title('Whole Table')

plt.xlabel('Index')

plt.ylabel('Value')

plt.legend(loc='upper right')

plt.show()

**Reading data from files and writing data to files**

Pandas is a powerful Python library for data manipulation and analysis. It provides various functions and methods for reading and writing data from/to files using different data structures like DataFrames and Series. Here's how you can read and write data using pandas:

**Reading Data from Files:**

1. **CSV Files:** CSV (Comma-Separated Values) files are one of the most common file formats for storing tabular data.

import pandas as pd

# Reading CSV file into a DataFrame

data = pd.read\_csv('data.csv')

# Display the DataFrame

print(data)

**Excel Files:** Pandas can also read data from Excel files (.xls or .xlsx).

import pandas as pd

# Reading Excel file into a DataFrame

data = pd.read\_excel('data.xlsx', sheet\_name='Sheet1')

# Display the DataFrame

print(data)

**Other Formats:** Pandas supports reading data from various other formats such as JSON, SQL databases, HTML tables, and more.

import pandas as pd

# Reading JSON file into a DataFrame

data = pd.read\_json('data.json')

# Display the DataFrame

print(data)

**Writing Data to Files:**

**1. CSV Files:**

import pandas as pd

# Creating a DataFrame

data = pd.DataFrame({

'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 28]

})

# Writing DataFrame to CSV file

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

Excel Files:

import pandas as pd

# Creating a DataFrame

data = pd.DataFrame({

'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 28]

})

**# Writing DataFrame to Excel file**

data.to\_excel('output.xlsx', sheet\_name='Sheet1', index=False)

**Other Formats:**

import pandas as pd

# Creating a DataFrame

data = pd.DataFrame({

'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 28]

})

**# Writing DataFrame to JSON file**

data.to\_json('output.json', orient='records')

**# Writing DataFrame to HTML file**

data.to\_html('output.html', index=False)

**Write a program to purposefully raise Indentation Error and Correct it**

program that intentionally raises an IndentationError and then corrects it:

def raise\_indentation\_error():

print("This will raise an IndentationError")

print("Because there's no indentation before the 'print' statements")

def correct\_indentation\_error():

print("This corrects the IndentationError")

print("By adding proper indentation before the 'print' statements")

# Uncomment the following lines to test the code

# raise\_indentation\_error()

# correct\_indentation\_error()

**Write a program to compute distance between two points taking input from the user**

# Python Program to Calculate Distance

# Reading co-ordinates

x1 = float(input('Enter x1: '))

y1 = float(input('Enter y1: '))

x2 = float(input('Enter x2: '))

y2 = float(input('Enter y2: '))

# Calculating distance

d = ( (x2-x1)\*\*2 + (y2-y1)\*\*2 ) \*\* 0.5

# Displaying result

print('Distance = %f' %(d))

**Program to display the following information: Your name, Full Address, Mobile Number, College Name, Course Subjects in python**

def personal\_details():

name, age = "Simon", 19

address = "Bangalore, Karnataka, India"

print("Name: {}\nAge: {}\nAddress: {}".format(name, age, address))

personal\_details()

**Write a Program for checking whether the given number is a even number or not.**

num = int(input("Enter a Number:"))

if num % 2 == 0:

print("Given number is Even")

else:

print("Given number is Odd")

**Program to find the largest three integers using if-else**

# Python program to find the largest number among the three input numbers

# change the values of num1, num2 and num3

# for a different result

num1 = 10

num2 = 14

num3 = 12

# uncomment following lines to take three numbers from user

#num1 = float(input("Enter first number: "))

#num2 = float(input("Enter second number: "))

#num3 = float(input("Enter third number: "))

if (num1 >= num2) and (num1 >= num3):

largest = num1

elif (num2 >= num1) and (num2 >= num3):

largest = num2

else:

largest = num3

print("The largest number is", largest)

**Python program to multiply two matrices**

# Program to multiply two matrices using list comprehension

# take a 3x3 matrix

A = [[12, 7, 3],

[4, 5, 6],

[7, 8, 9]]

# take a 3x4 matrix

B = [[5, 8, 1, 2],

[6, 7, 3, 0],

[4, 5, 9, 1]]

# result will be 3x4

result = [[sum(a \* b for a, b in zip(A\_row, B\_col))

for B\_col in zip(\*B)]

for A\_row in A]

for r in result:

print(r)

**Python Program to Find the Gcd of Two Numbers**

# Python code to demonstrate the working of gcd()

# importing "math" for mathematical operations

import math

# prints 12

print("The gcd of 60 and 48 is : ", end="")

print(math.gcd(60, 48))

To find the factorial of positive integer

**# Python program to find the factorial of a number provided by the user.**

# change the value for a different result

num = 7

# To take input from the user

#num = int(input("Enter a number: "))

factorial = 1

# check if the number is negative, positive or zero

if num < 0:

print("Sorry, factorial does not exist for negative numbers")

elif num == 0:

print("The factorial of 0 is 1")

else:

for i in range(1,num + 1):

factorial = factorial\*i

print("The factorial of",num,"is",factorial)

**Write a program To display prime number from 2 to n.**

1. lower\_value = int(input ("Please, Enter the Lowest Range Value: "))

2. upper\_value = int(input ("Please, Enter the Upper Range Value: "))

3.

4. print ("The Prime Numbers in the range are: ")

5. for number in range (lower\_value, upper\_value + 1):

6. if number > 1:

7. for i in range (2, number):

8. if (number % i) == 0:

9. break

10. else:

11. print (number)

**Program to write a series of random numbers in a file from 1 to n and display.**

import random

print("Random integers between 0 and 9: ")

for i in range(4, 15):

y = random.randrange(9)

print(y)

**Program to display a list of all unique words in a textfile**

text\_file = open('data.txt', 'r')

text = text\_file.read()

#cleaning

text = text.lower()

words = text.split()

words = [word.strip('.,!;()[]') for word in words]

words = [word.replace("'s", '') for word in words]

#finding unique

unique = []

for word in words:

if word not in unique:

unique.append(word)

#sort

unique.sort()

#print

print(unique)

**Write a program combine lists that combines these lists in to a dictionary**

# Quick examples of converting two lists into a dictionary

# Lists for keys and values

keys\_list = ['a', 'b', 'c']

values\_list = [1, 2, 3]

# Method 1: Using the zip() function and the dict() constructor

my\_dict = dict(zip(keys\_list, values\_list))

print(my\_dict)

# Method 2: Using the enumerate() function and a loop

my\_dict = {}

for i, key in enumerate(keys\_list):

my\_dict[key] = values\_list[i]

print(my\_dict)

# Method 3: Using the zip() function and a loop

my\_dict = {}

for k, v in zip(keys\_list, values\_list):

my\_dict[k] = v

print(my\_dict)

# Method 4: Using the fromkeys() method and a loop

my\_dict = dict.fromkeys(keys\_list, None)

for i in range(len(keys\_list)):

my\_dict[keys\_list[i]] = values\_list[i]

print(my\_dict)

**Python Inheritance**

class Animal:

# attribute and method of the parent class

name = ""

def eat(self):

print("I can eat")

# inherit from Animal

class Dog(Animal):

# new method in subclass

def display(self):

# access name attribute of superclass using self

print("My name is ", self.name)

# create an object of the subclass

labrador = Dog()

# access superclass attribute and method

labrador.name = "Rohu"

labrador.eat()

# call subclass method

labrador.display()

**Polymorphism**

class Cat:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def info(self):

print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")

def make\_sound(self):

print("Meow")

class Dog:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def info(self):

print(f"I am a dog. My name is {self.name}. I am {self.age} years old.")

def make\_sound(self):

print("Bark")

cat1 = Cat("Kitty", 2.5)

dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):

animal.make\_sound()

animal.info()

animal.make\_sound()

**Python program to Data visualization through Sea born for the above program 9.**

# Importing libraries

import numpy as np

import seaborn as sns

# Selecting style as white,

# dark, whitegrid, darkgrid

# or ticks

sns.set( style = "white" )

# Generate a random univariate

# dataset

rs = np.random.RandomState( 10 )

d = rs.normal( size = 50 )

# Plot a simple histogram and kde

# with binsize determined automatically

sns.distplot(d, kde = True, color = "g")

**Python program to Create an array of 10 zeros**

import numpy as np

array=np.zeros(10)

print("An array of 10 zeros:")

print(array)

array=np.ones(10)

print("An array of 10 ones:")

print(array)

array=np.ones(10)\*5

print("An array of 10 fives:")

print(array)

**Generate an array of 25 random numbers sampled from a standard normal distribution.**

import numpy as np

rand\_num = np.random.normal(0,1,25)

print("15 random numbers from a standard normal distribution:")

print(rand\_num)