Q1) Describe the functionality of the `describe()` function and its variations.

Ans) The **describe()** function and its variations are commonly associated with the Pandas library, which is widely used for data manipulation and analysis. The **describe()** function generates descriptive statistics that summarize the central tendency, dispersion, and shape of a dataset's distribution, excluding NaN values. Here's an overview of its functionality:

1)describe(): This function provides summary statistics of numeric columns by default. It returns a DataFrame that includes count, mean, standard deviation, minimum, 25th percentile (first quartile), median (50th percentile), 75th percentile (third quartile), and maximum values.

```
import pandas as pd



# Create a DataFrame

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

df = pd.DataFrame(data)



# Get summary statistics

summary = df.describe()

print(summary)
```

2)describe(include=['object']): This variation provides summary statistics for object (string) columns. It includes count, unique, top (most common value), and freq (frequency of top value).

```
# Add a new column with strings

df['C'] = ['a', 'b', 'c', 'a', 'a']
```

```
# Get summary statistics for object columns

summary_object = df.describe(include=['object'])

print(summary_object)
```

3)describe(include='all'): This variation includes summary statistics for both numeric and object columns.

##code
# Get summary statistics for all columns
```
summary_all = df.describe(include='all')

print(summary_all)
```

These variations of the **describe()** function are powerful tools for quickly understanding the characteristics of a dataset. They provide insights into the distribution of data, presence of outliers, and the general structure of the DataFrame.

Q2) Specify the maximum character limit permissible for file names and file paths in the Windows operating system. Additionally, enumerate the special characters that are prohibited in file names.

Ans) In Windows, the maximum length for a full file path, including the file name and extension, is 260 characters, and the maximum length for a file name is 255 characters.

Q 4) Identify alternatives to the Itertools module in Python.

Ans)Some alternate to Itertools module :

1)zip() Function: The **zip()** function pairs elements from multiple iterables.

```
# Example: Pair elements from two lists
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
pairs = zip(list1, list2)
```

2) **`enumerate()` Function:** The **`enumerate()`** function adds a counter to an iterable.

```
# Example: Enumerate elements in a list
my_list = ['a', 'b', 'c']
enumerated_list = enumerate(my_list)
```

3) **List Comprehensions:** List comprehensions can replace several functions from **`itertools`**, such as **`filter()`** and **`map()`**, in a more concise manner.

```
# Example: Filter even numbers using list comprehension
numbers = [1, 2, 3, 4, 5]
even_numbers = [x for x in numbers if x % 2 == 0]
```

4) **`functools.reduce()` Function:** The **`functools.reduce()`** function performs a rolling computation on an iterable.

```
# Example: Compute the product of elements in a list
from functools import reduce
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
```

5) **Custom Generators:** For more complex iterator logic, you can create custom generator functions.

```
# Example: Generate Fibonacci numbers using a generator function
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

6. Develop a module and associated packages within a library, followed by installation using pip.

Ans)

```
1) Create the project structure:
Start by creating a directory structure for your library. For example:

├── my_math/
│   ├── __init__.py
│   ├── addition.py
│   ├── subtraction.py
│   └── multiplication.py
├── README.md
└── setup.py

2)Write Code:

addition.py:
def add(x, y):
    return x + y
subtraction.py:
def subtract(x, y):
    return x - y
multiplication.py:
def multiply(x, y):
    return x * y
division.py:
def division(x, y):
    return x // y

3)Create setup.py:
from setuptools import setup, find_packages

setup(
    name='my_math_library',
    version='0.1',
    packages=find_packages(),
    install_requires=[],
    python_requires='>=3.6',
    author='Your Name',
    author_email='your@email.com',
    description='A simple math library',
    url='https://github.com/yourusername/my_math_library',
)
```

```
4)Package the Library:
Run the following command in the terminal to create a distribution package of
your library:

python setup.py sdist

5)Install Using pip:
After creating the distribution package, you can install it using pip:

pip install /path/to/my_math_library/dist/my_math_library-0.1.tar.gz

6)Usage:
Now you can use the functions from your library in your Python scripts or
interpreter:

from my_math.addition import add
from my_math.subtraction import subtract
from my_math.multiplication import multiply

print(add(5, 3))  # Output: 8
print(subtract(10, 4))  # Output: 6
print(multiply(2, 3))  # Output: 6
```

7. Instruct on the process of extracting date-time data from a DataFrame and exporting it to an Excel file using Pandas while preserving the original data type.

Ans)

```
import pandas as pd

# Example DataFrame with datetime column
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'DateOfBirth': ['1990-01-01', '1985-05-15', '1995-11-30']
}
df = pd.DataFrame(data)

# Convert 'DateOfBirth' column to datetime
df['DateOfBirth'] = pd.to_datetime(df['DateOfBirth'])

# Export datetime data to Excel while preserving data type
excel_file = 'datetime_data.xlsx'
```

```
with pd.ExcelWriter(excel_file, engine='xlsxwriter') as writer:
    df.to_excel(writer, index=False)
    writer.save()

print("Data exported to Excel file:", excel_file)
```

8. Propose alternative methods to replace the usage of the iloc function in Python.

Ans)

```
import pandas as pd

# create a sample dataframe
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
#Method 1: Using loc If you're using iloc to access rows and columns by label,
you can replace it with loc.
# using iloc
print(df.iloc[0, 0])  # output: 1

# using loc
print(df.loc[0, 'A'])  # output: 1

#Method 2: Using ix (deprecated) Note that ix is deprecated since pandas 0.20.0,
so it's not recommended to use it. However, I'll mention it for completeness.
print(df.ix[0, 'A'])  # output: 1

#Method 3: Using at or iat If you need to access a single value, you can use at
or iat. at is label-based, while iat is integer-based.
print(df.at[0, 'A'])  # output: 1
print(df.iat[0, 0])  # output: 1

#Method 4: Using query If you need to filter rows based on a condition, you can
use query.
print(df.query('A > 1'))  # output:   A  B
                         # 1  2  5
                         # 2  3  6
```

9. Illustrate the procedure for resetting the index in Python.

```
#Q.10)Illustrate the procedure for resetting the index in Python.
```

```python
import pandas as pd

# Sample DataFrame with custom index
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data, index=['a', 'b', 'c'])

# Display original DataFrame
print("Original DataFrame:")
print(df)

# Reset the index
df_reset = df.reset_index(drop=True)

# Display DataFrame after resetting the index
print("\nDataFrame after resetting index:")
print(df_reset)
```

Q3) Demonstrate how to achieve a matplotlib operation using plotly.

```python
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import numpy as np

x_data = [1, 2, 3, 4, 5]
y1_data = [10, 20, 30, 22, 12]
y2_data = [1, 4, 9, 16, 25]
y3_data = [25, 49, 11, 23, 35]
y4_data = [20, 21, 31, 24, 27]

fig1 = make_subplots(rows=1, cols=2)

trace1 = go.Scatter(x=x_data, y=y1_data, name="Plot A", mode='lines+markers',
marker=dict(color='red'))
fig1.add_trace(trace1, row=1, col=1)

trace2 = go.Scatter(x=x_data, y=y2_data, name="Plot B", mode='lines+markers',
marker=dict(color='orange'))
```

```
fig1.update_yaxes(title_text="Y-axis Label for Plot B", secondary_y=True, row=1,
col=1)
fig1.add_trace(trace2, row=1, col=1)

fig1.update_yaxes(range=[0, 50], row=1, col=1)

trace3 = go.Scatter(x=x_data, y=y3_data, name="Plot C", mode='lines+markers',
marker=dict(color='yellow'))
fig1.add_trace(trace3, row=1, col=2)

trace4 = go.Scatter(x=x_data, y=y4_data, name="Plot D", mode='lines+markers',
marker=dict(color='green'))
fig1.update_yaxes(title_text="Y-axis Label for Plot D", secondary_y=True, row=1,
col=2)
fig1.add_trace(trace4, row=1, col=2)

fig1.update_yaxes(range=[0, 50], row=1, col=2)

fig1.update_xaxes(title_text="X-axis Label")

fig1.update_layout(
    title="Combined Plots",
    legend_title_text="Legend",
    legend_orientation="v",  # Set legend orientation to 'v' for vertical
    legend=dict(x=0.02, y=1),
)

fig1.show()

fig2 = make_subplots(rows=1, cols=1)

trace1 = go.Scatter(x=x_data, y=y1_data, name="Plot A", mode='lines+markers',
marker=dict(color='red'))
fig2.add_trace(trace1, row=1, col=1)

fig2.update_yaxes(title_text="Y-axis Label for Plot B", secondary_y=True, row=1,
col=1)
fig2.add_trace(trace2, row=1, col=1)

trace3 = go.Scatter(x=x_data, y=y3_data, name="Plot C", mode='lines+markers',
marker=dict(color='yellow'))
fig2.add_trace(trace3, row=1, col=1)
```

```
trace4 = go.Scatter(x=x_data, y=y4_data, name="Plot D", mode='lines+markers',
marker=dict(color='green'))
fig2.add_trace(trace4, row=1, col=1)

fig2.update_yaxes(range=[0, 50], row=1, col=1)

fig2.update_xaxes(title_text="X-axis Label")

fig2.update_layout(
    title="Combined Plots",
    legend_title_text="Legend",
    legend_orientation="v",  # Set legend orientation to 'v' for vertical
    legend=dict(x=0.02, y=1),
)

fig2.show()
```

Q5) Provide a Python script demonstrating database connectivity, CSV loading, CRUD operations, and encoding handling between CSV (utf-8) and a database (latin-9). Furthermore, ensure the CSV dataset contains over 100,000 rows, encompassing words from diverse languages.

Ans)

```
import configparser
import pandas as pd
import pyodbc
import argparse
import time
config = configparser.ConfigParser()
config.read('config.ini')
def create_connection(server, database, username, password):
    conn = None
    try:
        conn = pyodbc.connect('DRIVER={SQL Server};SERVER=' + server +
                              ';DATABASE=' + database +
                              ';UID=' + username +
                              ';PWD=' + password)
        print("Database connection established.")
    except pyodbc.Error as e:
        print(e)
    return conn
def create_table_from_dataframe(conn, df, table_name):
    try:
        cursor = conn.cursor()

        # Drop the table if it exists
        drop_table_sql = f"DROP TABLE IF EXISTS {table_name}"
        cursor.execute(drop_table_sql)

        # Generate SQL column definitions dynamically based on DataFrame column
types
        columns_sql = ', '.join([f'[{col}] NVARCHAR(MAX)' for col in df.columns])

        # Create the table with UTF-8 collation
        create_table_sql = f"CREATE TABLE {table_name} ({columns_sql})"
        cursor.execute(create_table_sql)
        conn.commit()
        print(f"Table '{table_name}' created successfully based on DataFrame
schema.")
    except pyodbc.Error as e:
        print(e)
def bulk_insert(conn, target_table, data_file, encoding='utf-8',
field_terminator=',', row_terminator='\n', batch_size=None, max_errors=None):
    try:
        cursor = conn.cursor()
        # Construct the BULK INSERT statement
        bulk_insert_query = f"""
            BULK INSERT {target_table}
            FROM '{data_file}'
            WITH (
                DATAFILETYPE = 'char',
                CODEPAGE = '65001',   -- UTF-8 encoding
                FIELDTERMINATOR = '{field_terminator}',
                ROWTERMINATOR = '{row_terminator}',
                FIRSTROW = 2
                {', BATCHSIZE = ' + str(batch_size) if batch_size is not None
else ''}  -- Only add BATCHSIZE if not None
                {', MAXERRORS = ' + str(max_errors) if max_errors is not None
else ''}  -- Only add MAXERRORS if not None
            )
        """
        # Execute the BULK INSERT statement
        cursor.execute(bulk_insert_query)
        # Commit the transaction
        conn.commit()
        print("Data loaded from DataFrame to database successfully.")
    except pyodbc.Error as e:
        print(e)
        conn.rollback()



def main():
    parser = argparse.ArgumentParser(description="Database CRUD operations with
age_gender movie data")
    parser.add_argument("--config-file", type=str, help="Config file containing
database connection information")
    parser.add_argument("--csv-file", type=str, help="CSV file containing
age_gender data")
    parser.add_argument("--create-table", type=str, help="create table table-
name")
    args = parser.parse_args()
    df = pd.read_csv(args.csv_file, encoding='utf-8')  # Ensure CSV file is read
with UTF-8 encoding
    username = config.get('Settings','username')
    password = config.get('Settings','password')
    driver_name = config.get('Settings','DRIVER_NAME')
    database_name = config.get('Settings','DATABASE_NAME')
    server_name = config.get('Settings','SERVER_NAME')
    conn = create_connection(server_name, database_name, username, password)
    if conn is not None:
        table_name = args.create_table
        create_table_from_dataframe(conn, df, table_name=table_name)
        # Check if table creation was successful before proceeding with bulk
insert
        if table_name in [x.table_name for x in conn.cursor().tables()]:
            start_time = time.time()
            print(args.csv_file)
            filepath = args.csv_file
            bulk_insert(conn, target_table=table_name, data_file=filepath)
            time_taken = time.time() - start_time
            print("time_taken ", time_taken)
        else:
            print(f"Table '{table_name}' was not created successfully. Aborting
bulk insert.")
        conn.close()
```

```python
        print("Database connection closed.")
if __name__ == '__main__':
    main()
```

```
python q1.py --config-file config.ini --csv-file D:\output4.csv --create-table word
```