pandas

- How to Create a Data Frame with Fruits and Colors Example

Loading and Handling Datasets in Pandas

Pandas doesn't come with built-in datasets like some other libraries, but it offers many ways to load and handle external datasets. You can easily read data from CSV, Excel, SQL, JSON, and other formats using Pandas.

Here are common datasets you can load and work with in Pandas, along with some examples of reading them into your environment:

1. Loading CSV Datasets

You can load CSV files from your local system or directly from a URL into Pandas using pd.read_csv().

Example Titanic Dataset (from a URL)

```
import pandas as pd

# Loading the Titanic dataset from a URL
url =
"https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"
titanic = pd.read_csv(url)
print(titanic.head())
```

Example: Local CSV File

Loading a local CSV file

```
titanic = pd.read_csv("path_to_your_file/titanic.csv")
print(titanic.head())
```

2. Loading Excel Files

Pandas can easily read Excel files using pd.read_excel().

#xample: Superstore Dataset

Loading an Excel file

```
superstore = pd.read_excel("path_to_your_file/superstore_sales.xlsx")
print(superstore.head())
```

3. Loading JSON Files

You can load JSON files using pd.read_json().

Example: JSON File Loading

```
# Loading a JSON file
json_data = pd.read_json("path_to_your_file/data.json")
print(json_data.head())
```

4. Loading SQL Databases

If you're working with databases, Pandas can directly query them using SQL queries.

Example: Loading Data from SQL

```
import sqlite3

# Create connection to your SQLite database
conn = sqlite3.connect('database_name.db')

# Query the database
data = pd.read_sql_query("SELECT * FROM table_name", conn)
print(data.head())
```

5. Loading HTML Tables

Pandas can extract tables from HTML web pages using pd.read_html().

Example: Loading Data from an HTML Table

```
# Loading data from a webpage with HTML tables
url = "https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)"
tables = pd.read_html(url)
print(tables[0].head())  # The first table on the page
```

6. Loading Data from APIs

You can load data from APIs that return JSON, CSV, or other formats. For example, using the Kaggle API, you can download datasets and load them into Pandas.

Example: Loading Kaggle Dataset (after downloading)

```
# After downloading a dataset from Kaggle
kaggle_data = pd.read_csv("path_to_downloaded_kaggle_file.csv")
print(kaggle_data.head())
```

7. Loading Data from Google Sheets

You can also read data from Google Sheets by exporting them as CSV and reading into Pandas.

Example: Loading Data from Google Sheets

```
# Google Sheets shared link with export format as CSV
sheet_url = "https://docs.google.com/spreadsheets/d/your_sheet_id/export?
format=csv"
google_sheets_data = pd.read_csv(sheet_url)
print(google_sheets_data.head())
```

8. Loading Data from Zip Files

Pandas can read CSVs from zipped files directly without unzipping them.

Example: Loading from a Zip File

```
# Loading CSV from a zipped file
zip_url = "https://your_url/file.zip"
zipped_data = pd.read_csv(zip_url, compression='zip')
print(zipped_data.head())
```

9. Loading Data from a Clipboard

You can even copy data from somewhere and paste it into Pandas using pd.read_clipboard().

Example: Loading Clipboard Data

```
# Assuming you've copied a table from a webpage or a document
clipboard_data = pd.read_clipboard()
print(clipboard_data.head())
```

10. Sample Datasets in Python Libraries

While Pandas itself doesn’t provide built-in datasets, you can use datasets from libraries like Seaborn and Scikit-learn and load them into Pandas:

Example: Seaborn's Titanic Dataset into Pandas

```
import seaborn as sns

# Load Titanic dataset from Seaborn and convert to Pandas DataFrame
titanic = sns.load_dataset('titanic')
print(titanic.head())

Example: Scikit-learn Iris Dataset into Pandas

from sklearn.datasets import load_iris

# Load Iris dataset and convert to Pandas DataFrame
iris = load_iris()
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
print(iris_df.head())
```

How to store MySQL results in a pandas DataFrame using Python

There are two primary ways to store MySQL results in a pandas DataFrame using Python:

1. Using pandas.read_sql()

This is the recommended approach as it's specifically designed for this purpose. Here's how it works:

```
import pandas as pd
import mysql.connector

# Establish connection
mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)

# Define your SQL query
sql = "SELECT * FROM mytable"  # Replace with your specific query

# Read the results into a DataFrame
df = pd.read_sql(sql, mydb)  # mydb is the connection object

# Close the connection
mydb.close()

# Now you can work with the data in your DataFrame (df)
print(df.head())  # View the first few rows
```

Explanation:

- Import `pandas` and `mysql.connector`.
- Establish a connection to your MySQL database.
- Define your SQL query string (`sql`).
- Use `pd.read_sql(sql, mydb)` to execute the query and store the results in a pandas DataFrame named `df`. The `mydb` argument provides the connection object.
- Close the connection after reading the data.
- Now you can use the `df` DataFrame for further analysis or manipulation.

2. Using cursor.fetchall() and DataFrame constructor

This method involves fetching the results as a list of tuples and then constructing a DataFrame from it. Here's an example:

```
import pandas as pd
import mysql.connector

# Establish connection
mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)

mycursor = mydb.cursor()

# Execute your SQL query
sql = "SELECT * FROM mytable"
mycursor.execute(sql)

# Fetch the results
data = mycursor.fetchall()  # data is a list of tuples

# Define column names (optional, but recommended for clarity)
column_names = [i[0] for i in mycursor.description]  # Get column names from
cursor description

# Create the DataFrame
df = pd.DataFrame(data, columns=column_names)

# Close connection (same as previous method)
mycursor.close()
mydb.close()

# Now you can work with the DataFrame (df)
print(df.head())
```

Explanation:

- Import necessary libraries.

- Establish connection and create a cursor.
- Execute your SQL query using the cursor.
- Fetch the results using `fetchall()` which returns a list of tuples.
- Optionally, define column names based on the cursor description.
- Construct the DataFrame using `pd.DataFrame(data, columns=column_names)`.
- Close the connection.
- Now you can use the `df` DataFrame for further analysis.

Choosing the right approach:

- `pandas.read_sql()` is generally preferred as it's more concise and efficient, especially for larger datasets.
- The cursor-based approach might be useful if you need more control over the cursor object or want to perform additional operations before constructing the DataFrame.

Example #: Using SQLAlchemy Engine

```
from sqlalchemy import create_engine

# Construct the connection URL (replace with your credentials)
engine =
create_engine("mysql+mysqlconnector://yourusername:yourpassword@host/yourdatabase"
)

df = pd.read_sql(sql, engine)
```

```
from sqlalchemy import create_engine

# Construct the connection URL (replace with your credentials)
engine = create_engine("mysql+mysqlconnector://root:abc1234@localhost/library")

# Define your SQL query
sql = "SELECT * FROM books"  # Replace with your specific query

# Read the results into a DataFrame
df = pd.read_sql(sql, engine)  # mydb is the connection object

# Close the connection


# Now you can work with the data in your DataFrame (df)
print(df.head())  # View the first few rows
```

Example #: Using Database String URI

```
import pandas as pd
```

```
# Replace with your connection string details
connection_string =
"mysql+mysqlconnector://yourusername:yourpassword@host/yourdatabase"
df = pd.read_sql(sql, connection_string)
```