**Importing Data Sets**

In this module, you will learn how to understand data and learn about how to use the libraries in Python to help you import data from multiple sources. You will then learn how to perform some basic tasks to start exploring and analyzing the imported data set.

**Learning Objectives**

* Access databases using Python database APIs
* Analyze Python data using a dataset
* Identify three Python libraries and describe their uses
* Read data using Python's Pandas package
* Demonstrate how to import and export data in Python

# **Importing Data Sets**

## **Course Introduction**

* **Introduction to Data Science with Python**:
  + Importance of Python in data science.
  + Overview of libraries like NumPy, Pandas, and scikit-learn.
* **Module Breakdown**:
  + **Module 1**: Understanding dataset characteristics, importing data, and starting analysis.
  + **Module 2**: Data wrangling, preprocessing, handling missing values, formatting, and normalization.
  + **Module 3**: Exploratory data analysis (EDA), descriptive statistics, GroupBy, and correlation.
  + **Module 4**: Linear regression, model evaluation, polynomial regression, and decision-making.
  + **Module 5**: Model evaluation and refinement, overfitting, underfitting, and grid search.
* **Hands-On Project**: Application of skills learned using a real-world dataset.

## **Understanding the Data**

* **Dataset Overview**:
  + The dataset is an open dataset by Jeffrey C Schlimmer in CSV format.
  + Each line represents a row in the dataset.
* **First Row**:
  + The first row is not a header; it contains data.
* **Attributes**:
  + There are **26 columns** in total.
  + **Symboling**: Indicates the insurance risk level of a car (values range from -3 to +3).
  + **Normalized Losses**: Represents the average loss payment per insured vehicle per year, normalized for car size (values range from 65 to 256).
  + The **26th attribute** is **price**, which is the target value we want to predict.
* **Goal**:
  + The objective is to predict the price based on other car features.
* **Note**:
  + The dataset is from **1985**, so car prices may seem low.

## **Python Packages for Data Science**

1. **Scientific Computing Libraries**
   * **Pandas**:
     + Data structure for data manipulation and analysis.
     + Uses **DataFrames** (2D tables with rows and columns).
   * **NumPy**:
     + Utilizes arrays for inputs/outputs.
     + Supports matrix operations and fast array processing.
   * **SciPy**:
     + Advanced math functions and data visualization.
2. **Data Visualization Libraries**
   * **Matplotlib**:
     + Widely used for creating graphs and plots.
     + Highly customizable.
   * **Seaborn**:
     + Built on Matplotlib.
     + Simplifies the creation of complex visualizations (e.g., heat maps, violin plots).
3. **Machine Learning Libraries**
   * **Scikit-learn**:
     + Tools for statistical modeling (regression, classification, clustering).
     + Built on NumPy, SciPy, and Matplotlib.
   * **Statsmodels**:
     + Explore data, estimate statistical models, and perform tests.

Learning Objectives

* Develop Python code for data cleaning and preparation.
* Perform exploratory data analysis using Pandas and NumPy.
* Manipulate data using DataFrames and summarize data.
* Build and evaluate regression models using Scikit-learn.

## **Importing and Exporting data in Python**

1. **Data Acquisition**:
   * Data acquisition involves loading and reading data into a notebook from various sources.
   * Two important factors to consider: **format** and **file path**.
2. **Data Formats**:
   * Common data formats include:
     + **CSV** (Comma-Separated Values)
     + **JSON**
     + **XLSX**
     + **HDF**
   * The format is often indicated by the file extension.
3. **Reading Data with Pandas**:
   * Use the read\_csv method to read CSV files into a Pandas DataFrame.
   * Example code:
   * import pandas as pd

df = pd.read\_csv('file\_path.csv', header=None)

* + **Header Handling**: If the data has no headers, set header=None.

1. **Inspecting Data**:
   * Use dataframe.head() to view the first few rows of the DataFrame.
   * Use dataframe.tail() to view the last few rows.
2. **Assigning Column Names**:
   * If you have column names in a separate list, you can assign them as follows:
   * headers = ['Column1', 'Column2', 'Column3']

df.columns = headers

1. **Exporting Data**:
   * To save a DataFrame to a new CSV file, use the to\_csv method:

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

Detailed Explanation of the Code

* **Importing Pandas**:

import pandas as pd

* + This line imports the Pandas library, which is essential for data manipulation and analysis in Python.
* **Reading a CSV File**:

df = pd.read\_csv('file\_path.csv', header=None)

* + pd.read\_csv() reads the CSV file located at 'file\_path.csv'.
  + The header=None argument indicates that the CSV file does not contain header rows, so Pandas will automatically assign integer indices as column names.
* **Inspecting the Data**:

print(df.head())

* + This command prints the first five rows of the DataFrame, allowing you to quickly check if the data was loaded correctly.
* **Assigning Column Names**:
* headers = ['Column1', 'Column2', 'Column3']

df.columns = headers

* + Here, you create a list of headers and assign it to the DataFrame's columns. This makes the DataFrame easier to work with.
* **Exporting the DataFrame**:

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

* + This line saves the DataFrame to a new CSV file named 'automobile.csv'.
  + The index=False argument prevents Pandas from writing row indices to the file.

Summary

* **Pandas** is a powerful library for data manipulation in Python.
* Understanding how to read, inspect, and export data is crucial for data analysis.
* Always check the format and structure of your data to ensure proper handling.

## **Getting Started Analyzing data in Python**

1. **Understanding Data Types in Pandas**:
   * **Main Data Types**:
     + **Object**: Similar to strings in Python.
     + **Float**: Represents continuous numeric values.
     + **Int**: Represents integer values.
     + **DateTime**: Useful for handling time series data.
   * **Importance of Data Types**:
     + Pandas automatically assigns data types, which may sometimes be incorrect.
     + Checking data types helps in applying appropriate functions to the data.
2. **Checking Data Types**:
   * Use the dtypes attribute to see the data type of each column in a DataFrame.
   * Example:
   * import pandas as pd
   * df = pd.read\_csv('data.csv')

print(df.dtypes)

1. **Statistical Summary**:
   * Use the describe() method to get a statistical summary of numerical columns.
   * This includes count, mean, standard deviation, min, max, and quartiles.
   * Example:
   * summary = df.describe()

print(summary)

1. **Including Object Type Columns**:
   * To include non-numeric columns in the summary, use:
   * summary = df.describe(include='all')

print(summary)

1. **Handling NaN Values**:
   * NaN stands for "Not a Number" and indicates missing values in the dataset.
2. **Concise Summary of DataFrame**:
   * Use the info() method to get a concise summary of the DataFrame, including index, data types, and memory usage.
   * Example:

df.info()

Explanation:

* Understanding data types is crucial for data analysis as it affects how you can manipulate and analyze the data. For instance, mathematical operations can only be performed on numerical data types.
* The describe() method provides a quick overview of the dataset, helping identify potential issues like outliers or incorrect data types.
* The info() method is useful for getting a quick summary of the DataFrame, which can help in understanding the structure of the dataset.

## **Accessing Databases with Python**

1. **Accessing Databases with Python**:
   * Python can connect to databases using API calls.
   * The connection is typically made through a library that implements the DB API.
2. **SQL API and Python DB API**:
   * **SQL API**: A set of functions that allows you to send SQL statements to a Database Management System (DBMS).
   * **DB API**: A standard interface for accessing relational databases in Python, allowing for code that works with multiple database types.
3. **Connection and Cursor Objects**:
   * **Connection Object**: Used to connect to the database and manage transactions.
   * **Cursor Object**: Used to execute queries and fetch results.
4. **Common Methods**:
   * cursor(): Returns a new cursor object.
   * commit(): Commits any pending transaction to the database.
   * rollback(): Rolls back to the start of any pending transaction.
   * close(): Closes the database connection.

Example Code Snippet:

Here’s a simple example of how to connect to a database using Python and the DB API:

import sqlite3 # Importing the SQLite library

# Establishing a connection to the database

connection = sqlite3.connect('example.db') # Replace 'example.db' with your database name

# Creating a cursor object

cursor = connection.cursor()

# Creating a table

cursor.execute('''CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT)''')

# Inserting data into the table

cursor.execute("INSERT INTO users (name) VALUES ('Alice')")

cursor.execute("INSERT INTO users (name) VALUES ('Bob')")

# Committing the transaction

connection.commit()

# Querying the data

cursor.execute("SELECT \* FROM users")

rows = cursor.fetchall() # Fetching all results

# Displaying the results

for row in rows:

print(row)

# Closing the connection

connection.close()

Explanation of the Code:

* **Importing the Library**: The sqlite3 library is imported to work with SQLite databases.
* **Connecting to the Database**: The connect() function establishes a connection to the specified database file.
* **Creating a Cursor**: The cursor() method creates a cursor object to execute SQL commands.
* **Creating a Table**: The execute() method runs a SQL command to create a table if it doesn't already exist.
* **Inserting Data**: Additional execute() calls insert data into the table.
* **Committing Changes**: The commit() method saves the changes made to the database.
* **Querying Data**: A SELECT statement retrieves data from the table, and fetchall() collects all results.
* **Displaying Results**: A loop prints each row retrieved from the database.
* **Closing the Connection**: Finally, the close() method is called to close the database connection.

**Lesson Summary**

Congratulations! You have completed this lesson. At this point in the course, you know:

* Each line in a dataset is a row, and commas separate the values.
* To understand the data, you must analyze the attributes for each column of data.
* Python libraries are collections of functions and methods that facilitate various functionalities without writing code from scratch and are categorized into Scientific Computing, Data Visualization, and Machine Learning Algorithms.
* Many data science libraries are interconnected; for instance, Scikit-learn is built on top of NumPy, SciPy, and Matplotlib.
* The data format and the file path are two key factors for reading data with Pandas.
* The **read\_CSV** method in Pandas can read files in CSV format into a Pandas DataFrame.
* Pandas has unique data types like object, float, Int, and datetime.
* Use the **dtype** method to check each column’s data type; misclassified data types might need manual correction.
* Knowing the correct data types helps apply appropriate Python functions to specific columns.
* Using **Statistical Summary** with **describe()** provides count, mean, standard deviation, min, max, and quartile ranges for numerical columns.
* You can also use **include='all'** as an argument to get summaries for object-type columns.
* The statistical summary helps identify potential issues like outliers needing further attention.
* Using the **info() Method** gives an overview of the top and bottom 30 rows of the DataFrame, useful for quick visual inspection.
* Some statistical metrics may return "NaN," indicating missing values, and the program can’t calculate statistics for that specific data type.
* Python can connect to databases through specialized code, often written in Jupyter notebooks.
* SQL Application Programming Interfaces (APIs) and Python DB APIs (most often used) facilitate the interaction between Python and the DBMS.
* **SQL APIs** connect to DBMS with one or more API calls, build SQL statements as a text string, and use API calls to send SQL statements to the DBMS and retrieve results and statuses.
* **DB-API**,Python's standard for interacting with relational databases,uses **connection objects** to establish and manage database connections and **cursor objects** to run queries and scroll through the results.
* Connection Object methods include the cursor(), commit(), rollback(), and close() commands.
* You can import the database module, use the **Connect API** to open a connection, and then create a cursor object to run queries and fetch results.
* Remember to close the database connection to free up resources.