Python

Unit- 4

Advanced Topics I: Data Science and Data Visualization using Python

Data Science Using Python: Data Frames

A **DataFrame** is a two-dimensional, table-like data structure in pandas, a powerful Python library for data manipulation and analysis. It is similar to an Excel spreadsheet or an SQL table, where data is stored in rows and columns.

---

1. Creating a DataFrame from an Excel Spreadsheet

To read an Excel file into a pandas DataFrame, we use the pandas.read_excel() function.

Example:

```
import pandas as pd

# Load data from an Excel file
df = pd.read_excel("data.xlsx")

# Display the first 5 rows
print(df.head())
```

Requirements: The openpyxl library is needed to read .xlsx files. Install it using:

pip install openpyxl

2. Creating a DataFrame from a .csv File

A CSV (Comma-Separated Values) file can be loaded into a DataFrame using pandas.read_csv().

Example:

```
import pandas as pd

# Load data from a CSV file
df = pd.read_csv("data.csv")

# Display the first 5 rows
print(df.head())
```

Notes:

- Ensure the file exists in the working directory.
- You can specify delimiters if needed, e.g., pd.read_csv("data.csv", delimiter=";").

3. Creating a DataFrame from a Python Dictionary

A dictionary where keys represent column names and values are lists can be converted into a DataFrame.

Example:

```
import pandas as pd

# Create a dictionary
data = {
   "Name": ["Ironman", "Devil", "Charlie"],
   "Age": [25, 30, 35],
   "City": ["New York", "Los Angeles", "Chicago"]
}

# Convert dictionary to DataFrame
df = pd.DataFrame(data)

# Display DataFrame
print(df)
```

4. Creating a DataFrame from a List of Tuples

A list of tuples can be converted into a DataFrame, where each tuple represents a row.

Example:

```
import pandas as pd

# List of tuples
data = [("Ironman", 25, "New York"),
        ("Lucifer", 30, "Los Angeles"),
        ("Charlie", 35, "Chicago")]

# Define column names
df = pd.DataFrame(data, columns=["Name", "Age", "City"])

# Display DataFrame
print(df)
```

5. Operations on DataFrames in Python (Using Pandas)

Once a DataFrame is created in pandas, various operations can be performed to manipulate and analyze the data effectively.

A. Display Basic Information

1. Display Summary of DataFrame (info())

```
print( df.info() )
```

Explanation:

- Displays column names, data types, and non-null value counts.
- Useful for checking missing values and memory usage.

2. Display Statistical Summary (describe())

```
print(df.describe())
```

Explanation:

- Provides statistical insights like count, mean, std (standard deviation), min, max, and percentiles for numerical columns.
- Helps in understanding data distribution.

B. Selecting Columns

You can select specific columns from a DataFrame.

1. Selecting a Single Column

```
print(df["Name"])
```

Explanation:

- Extracts a single column (Series object) from the DataFrame.
- Equivalent to df.Name (if column name doesn't have spaces).

2. Selecting Multiple Columns

```
print(df[["Name", "City"]])
```

Explanation:

- Extracts multiple columns as a new DataFrame.
- Column names should be provided as a list.

C. Filtering Data

Filtering is used to select specific rows based on conditions.

```
Example: Get rows where Age > 28

filtered_df = df[df["Age"] > 28]

print(filtered_df)
```

Explanation:

- Uses a boolean condition to filter rows.
- Returns a subset where the "Age" column is greater than 28.

D. Adding a New Column

You can add new columns to a DataFrame dynamically.

```
Example: Adding a "Salary" Column

df["Salary"] = [50000, 60000, 70000]

print(df)
```

Explanation:

- Creates a new column named "Salary" and assigns values to each row.
- The length of the new column values must match the number of rows.

E. Deleting a Column

Unnecessary columns can be removed using drop().

```
Example: Deleting the "Salary" Column

df.drop(columns=["Salary"], inplace=True)

print(df)
```

Explanation:

- Removes the specified column.
- inplace=True modifies the original DataFrame.

F. Sorting Data

Sorting helps in organizing data in ascending or descending order.

Example: Sorting by Age (Descending Order)

```
sorted_df = df.sort_values(by="Age", ascending=False)

print(sorted_df)
```

Explanation:

- Sorts the DataFrame by the "Age" column in descending order.
- Use ascending=True for ascending order.

G. Resetting Index

Sometimes, after filtering or sorting, the index numbers become unordered. Resetting the index can fix this.

Example: Reset Index

```
df.reset_index(drop=True, inplace=True)

print(df)
```

Explanation:

- Resets the index of the DataFrame.
- drop=True removes the old index column.

Data Visualization Using Python

Data visualization is a key aspect of data analysis. It helps in understanding patterns, trends, and insights from data. Python provides several libraries for visualization, but the most commonly used is **Matplotlib**.

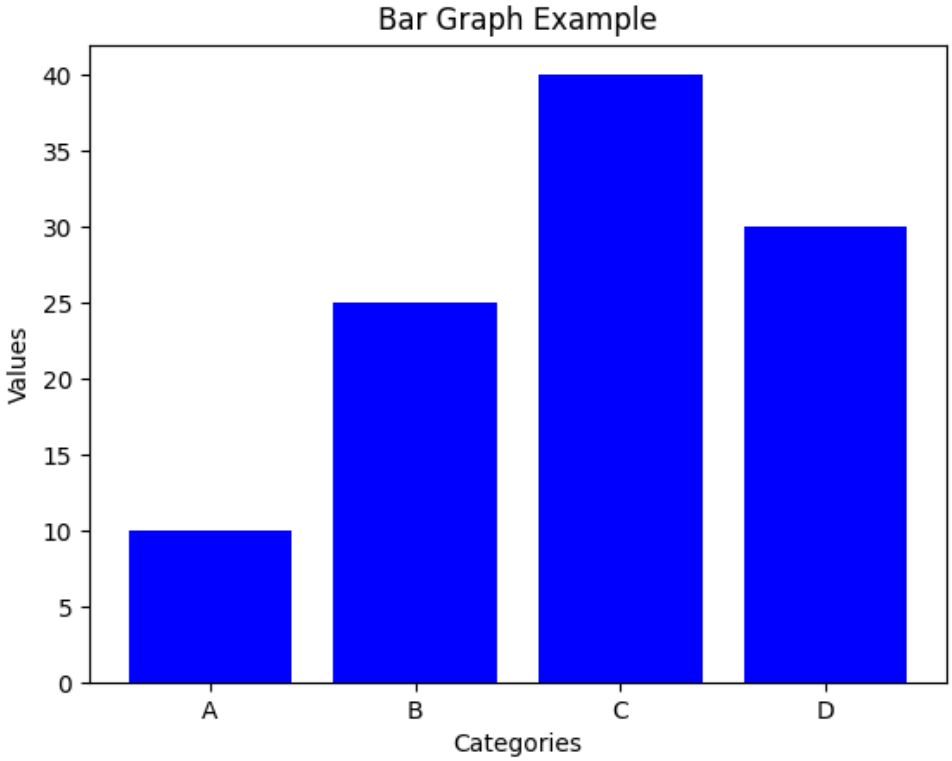
---

1. Bar Graph

A **bar graph** is used to represent categorical data using rectangular bars.

Example: Creating a Bar Graph

```
import matplotlib.pyplot as plt

# Data
categories = ['A', 'B', 'C', 'D']
values = [10, 25, 40, 30]

# Create bar graph
plt.bar(categories, values, color='blue')

# Labels and title
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Bar Graph Example")

# Show plot
plt.show()
```



Notes:

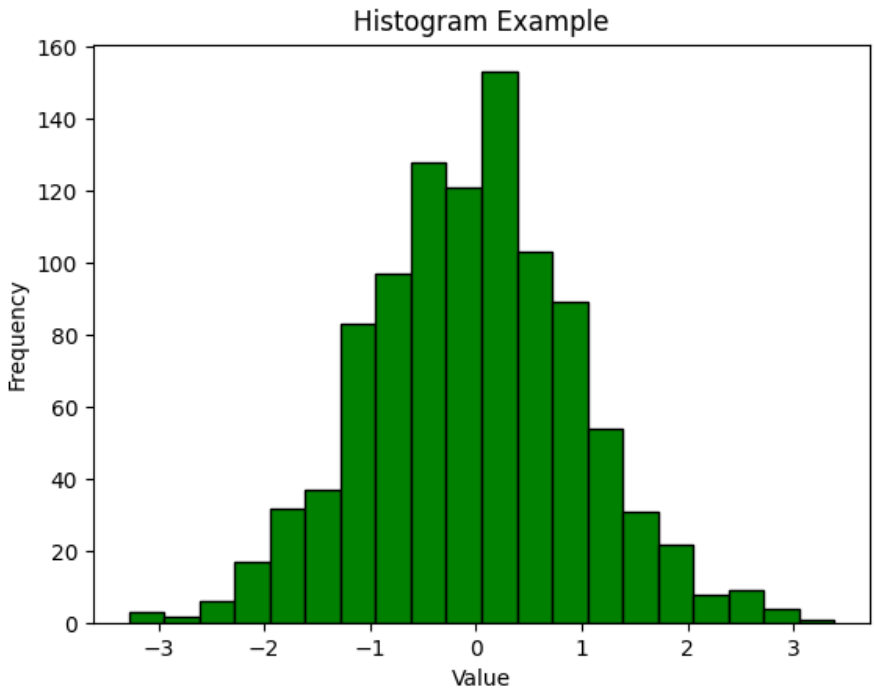
- plt.bar(x, y) creates a bar graph where x represents categories and y represents values.
- You can change the color, width, and alignment of bars.

2. Histogram

A **histogram** is used to visualize the distribution of continuous data.

Example: Creating a Histogram

```
import numpy as np
import matplotlib.pyplot as plt

# Generate random data
data = np.random.randn(1000)

# Create histogram
plt.hist(data, bins=20, color='green', edgecolor='black')

# Labels and title
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.title("Histogram Example")

# Show plot
plt.show()
```



Notes:

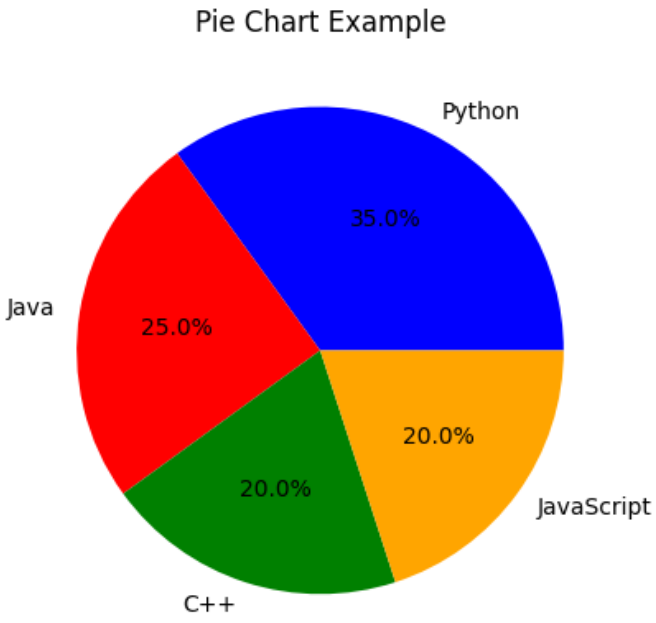
- plt.hist(data, bins=n) creates a histogram where bins define the number of intervals.
- The histogram groups continuous data into bins and counts occurrences.

3. Pie Chart

A **pie chart** represents data as slices of a circle, showing proportions.

Example: Creating a Pie Chart

```
# Data
labels = ['Python', 'Java', 'C++', 'JavaScript']
sizes = [35, 25, 20, 20]

# Create pie chart
plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=['blue', 'red', 'green', 'orange'])

# Title
plt.title("Pie Chart Example")

# Show plot
plt.show()
```



Notes:

- autopct='%1.1f%%' displays percentages.
- You can add explode = [0.1, 0, 0, 0] to highlight a slice.

4. Line Graph

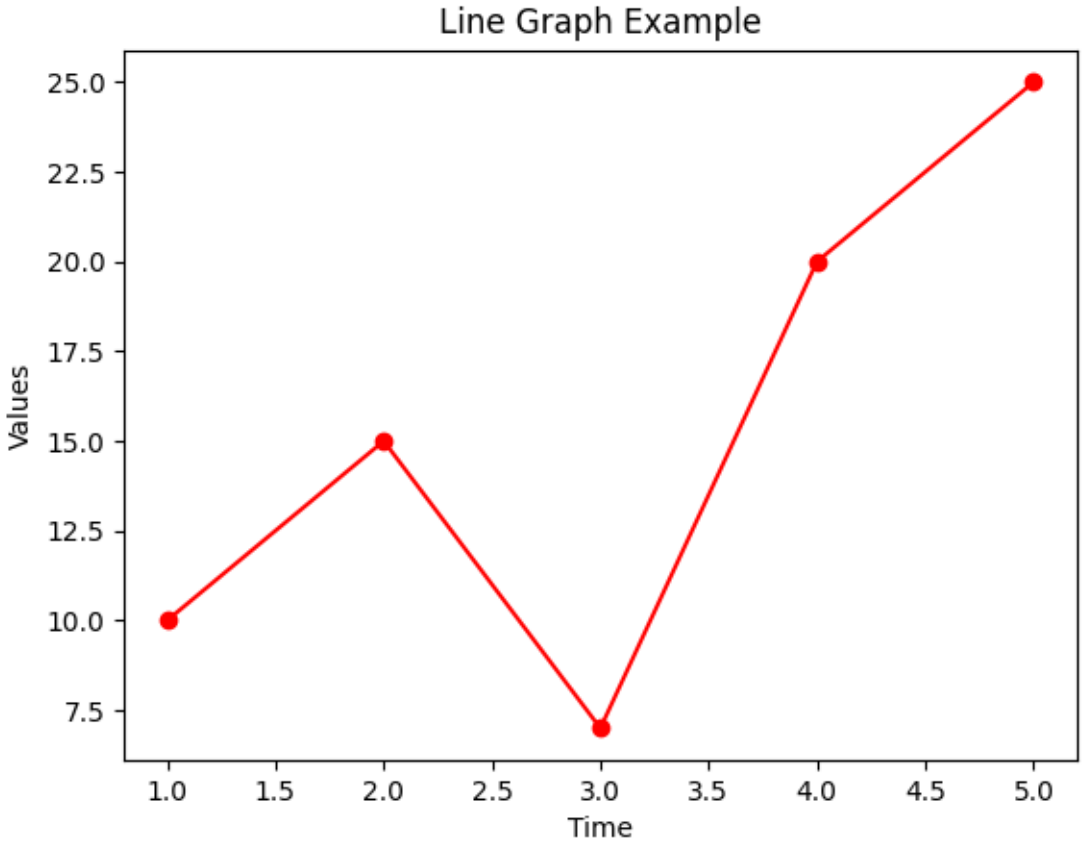
A **line graph** is useful for showing trends over time.

Example: Creating a Line Graph

```
# Data
x = [1, 2, 3, 4, 5]
y = [10, 15, 7, 20, 25]

# Create line graph
plt.plot(x, y, marker='o', linestyle='-', color='red')

# Labels and title
plt.xlabel("Time")
plt.ylabel("Values")
plt.title("Line Graph Example")

# Show plot
plt.show()
```

Notes:

- plt.plot(x, y, marker='o') adds markers to points.
- You can change linestyle='dashed' for a dashed line.



Use for

- **Bar Graph** → Categorical comparison
- **Histogram** → Data distribution
- **Pie Chart** → Proportions
- **Line Graph** → Trends over time

Plotting Using PyLab & Mortgage Visualization in Python

Plotting data is essential for data analysis, and **PyLab** (a module within Matplotlib) provides an easy way to create plots. Additionally, we will cover how to **visualize mortgage payments** with an extended example.

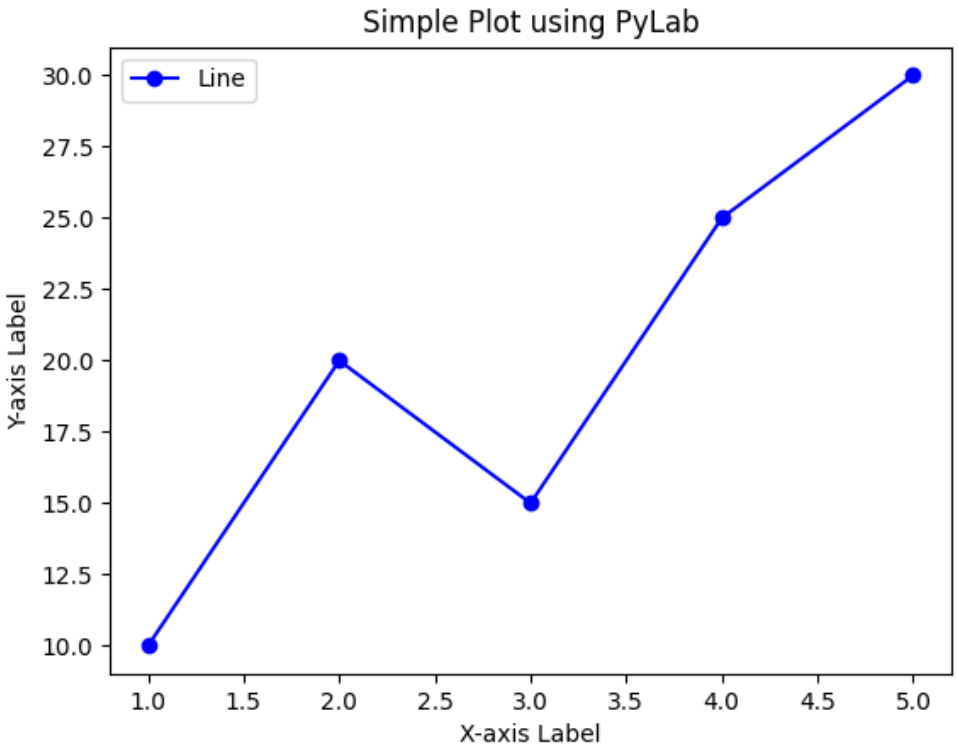
---

1. Plotting Using PyLab

PyLab is a module in Matplotlib that combines **NumPy and plotting functionalities**. You can use PyLab to create simple and advanced plots.

1.1 Basic Plot Using PyLab

```
from pylab import *

# Data

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

y = [10, 20, 15, 25, 30]


# Create a plot

plot(x, y, marker='o', linestyle='-', color='b', label="Line")


# Labels and title

xlabel("X-axis Label")

ylabel("Y-axis Label")

title("Simple Plot using PyLab")


# Show legend

legend()


# Display the plot

show()
```

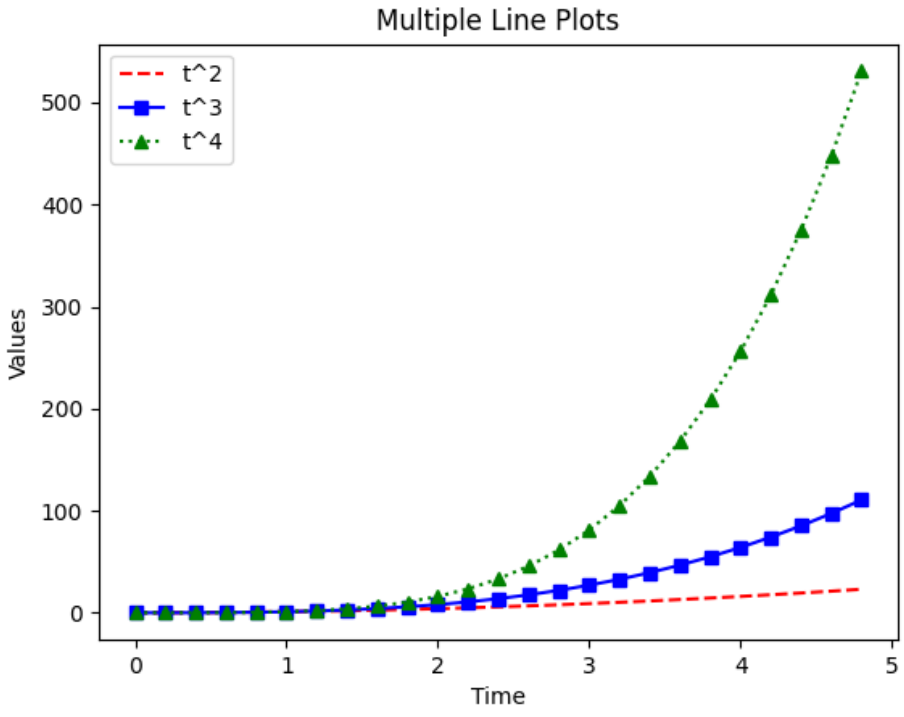


Explanation:

- plot(x, y) creates a line graph.
- xlabel(), ylabel(), and title() add descriptions.
- show() displays the plot.

1.2 Multiple Line Plots

```
from numpy import arange
from matplotlib.pyplot import xlabel, ylabel, title, legend, show, plot


t = arange(0., 5., 0.2)  # Generating time intervals
plot(t, t**2, 'r--', label="t^2")  # Red dashed line
plot(t, t**3, 'bs-', label="t^3")  # Blue squares
plot(t, t**4, 'g^:', label="t^4")  # Green triangles

# Labels and title
xlabel("Time")
ylabel("Values")
title("Multiple Line Plots")

legend() # Show legend
show() # Display the plot
```



Explanation:

- Uses **different styles** ('r--' = red dashed, 'bs-' = blue squares, 'g^:' = green triangles).
- legend() helps differentiate plots.

2. Plotting Mortgages & Financial Data

What is Mortgage?

A mortgage is a loan where you use your property as collateral to borrow money. The lender has the right to take your property if you don't repay the loan plus interest.

2.1 Understanding Mortgage Payments

A mortgage is a loan used to purchase a home. The monthly payment depends on:

- Loan Amount (Principal)
- Interest Rate
- Loan Duration (Years)

We can use Python to **visualize mortgage payments** over time.

2.2 Mortgage Payment Calculation

The formula for a **fixed monthly payment** is:

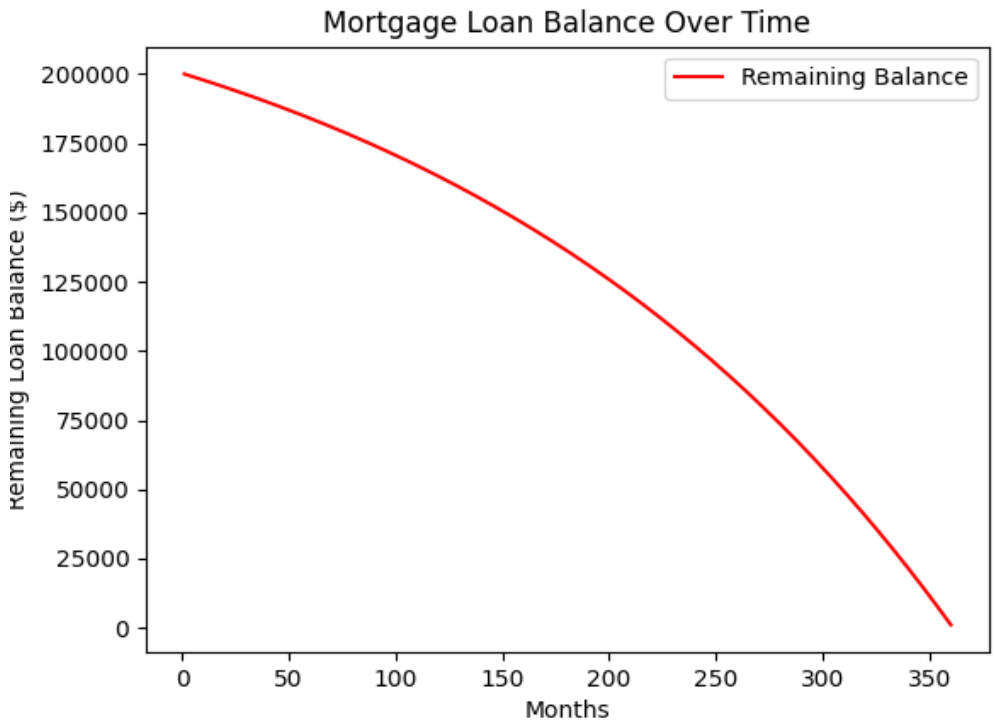
$$M = \frac{P \times r \times (1+r)^n}{(1+r)^n - 1}$$

Where:

- **M** = Monthly payment
- **P** = Loan amount (Principal)
- **r** = Monthly interest rate (Annual Rate / 12 / 100)
- **n** = Total number of payments (Years * 12)

2.3 Mortgage Visualization in Python

```
import numpy as np
import matplotlib.pyplot as plt

# Mortgage Loan Parameters
loan_amount = 200000   # Loan principal
annual_interest_rate = 5   # Annual interest rate (5%)
years = 30   # Loan term in years

# Convert interest rate to monthly
r = (annual_interest_rate / 100) / 12
n = years * 12  # Total payments

# Calculate fixed monthly payment
monthly_payment = (loan_amount * r * (1 + r) ** n) / ((1 + r) ** n - 1)

# Generate a payment schedule
months = np.arange(1, n + 1)
remaining_balance = [loan_amount]

for i in months[1:]:
    interest = remaining_balance[-1] * r
    principal = monthly_payment - interest
    remaining_balance.append(remaining_balance[-1] - principal)

# Plot mortgage balance over time
plt.plot(months, remaining_balance, label="Remaining Balance", color='red')

# Labels and title
plt.xlabel("Months")
plt.ylabel("Remaining Loan Balance ($)")
plt.title("Mortgage Loan Balance Over Time")
plt.legend()

plt.show()  # Display the plot
```



Explanation:

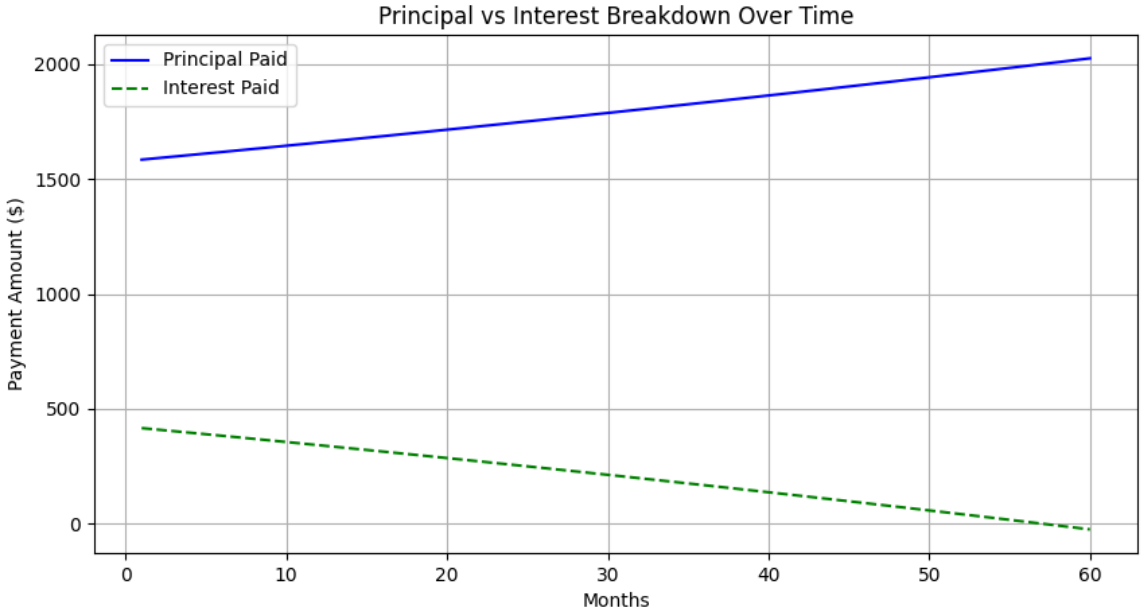
- The **remaining balance** decreases over time as payments are made.
- We separate **interest** and **principal** in each monthly payment.
- The red curve shows how much of the loan is left to pay.

3. Extended Mortgage Example: Interest vs. Principal Payments

Another way to analyze mortgage payments is by breaking down **how much goes to principal vs. interest** each month.

Example: Visualizing Principal & Interest Payments

```
# Initialize values
principal_paid = [ ]
interest_paid = [ ]
remaining_balance = loan_amount

for i in range(n):
   interest = remaining_balance * r
   principal = monthly_payment - interest
   remaining_balance -= principal

   interest_paid.append(interest)
   principal_paid.append(principal)

# Plot Principal vs. Interest Payments
plt.plot(months, principal_paid, label="Principal Paid", color='blue')
plt.plot(months, interest_paid, label="Interest Paid", color='green')

# Labels and title
plt.xlabel("Months")
plt.ylabel("Payment Amount ($)")
plt.title("Principal vs Interest Breakdown")
plt.legend()
plt.show()  # Display the plot
```



Interpretation:

- The **blue line (principal)** starts **low** but increases over time.
- The **green line (interest)** starts **high** but decreases over time.
- This shows how in the early years, **most of the payment goes to interest**, but over time, **more goes to the principal**.

PyLab provides a simple way to create plots in Python.

Mortgage visualization helps understand **loan payments, remaining balance, and breakdown of principal & interest**.