12 Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program. To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

In simple words, Python module is a file containing Python definitions and statements. A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module. Modules allow you to organize Python code logically into reusable files, making it easier to manage and maintain your codebase. Modules help in avoiding naming conflicts, provide code reusability, and enhance the overall structure of a Python program. Modules in Python provides us the flexibility to organize the code in a logical way. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized. Consider a module to be the same as a code library. A file containing a set of functions you want to include in your application.

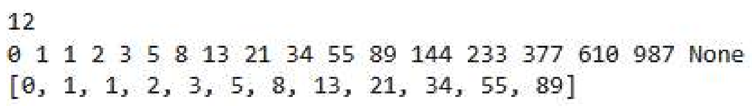
```
# Save this code in a file named mymodule.py
```

```
def greeting(name):
  print("Hello, " + name)
# A simple module, calc.py
def add(x, y):
    return (x+y)
def subtract(x, y):
    return (x-y)
 # Fibonacci numbers module
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

```
# Import the module named mymodule, and call the greeting function:
```

```
import mymodule
mymodule.greeting("Jonathan")
print(mymodule.add(10, 2))
print(mymodule.fib(1000))
print(mymodule.fib2(100))
```

```
Output:
```



```
12
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 None
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches for importing a module. For example, to import the module calc.py, we need to put the following command at the top of the script. Note: This does not import the functions or classes directly instead imports the module only. A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur. To access the functions inside the module the dot(.) operator is used.

12.1 Library and Packages

Any Python file is a module, its name being the file's base name without the .py extension. A package is a collection of Python modules: while a module is a single Python file, a package is a directory of Python modules containing an additional **init**.py file, to distinguish a package from a directory that just happens to contain a bunch of Python scripts. Packages can be nested to any depth, provided that the corresponding directories contain their own **init**.py file. When a module/package/something else is "published" people often refer to it as a library. Often libraries contain a package or multiple related packages, but it could be even a single module. Libraries usually do not provide any specific functionality, i.e. you cannot "run a library".

12.2 Advantages of Modules

Some of the advantages while working with modules in Python is as follows:

- Reusability. Working with modules makes the code reusable.
- Simplicity. The module focuses on a small proportion of the problem, rather than focusing on the entire problem.
- Scoping. A separate namespace is defined by a module that helps to avoid collisions between identifiers.

12.3 Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc. For Example, save this code in the file mymodule.py

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Now, import the module named mymodule, and access the person1 dictionary:

```
import mymodule
a = mymodule.person1["age"]
print(a)
```

12.4 Naming a Module

You can name the module file whatever you like, but it must have the file extension .py. You can create an alias when you import a module, by using the as keyword. For Example, create an alias for mymodule called mx:

```
import mymodule as mx
a = mx.person1["age"]
print(a)
```

12.5 Built-in Modules

There are several built-in modules in Python, which you can import whenever you like. For Example, Import and use the platform module:

```
import platform
x = platform.system()
print(x)
```

12.6 The from import Statement

Python’s from statement lets you import specific attributes from a module without importing the module as a whole.

NOTE: Here we have to note that when we try to import using the from keyword, then do not use the module name when referring to elements in the module.

```
# importing sqrt() and factorial from the module math
from math import sqrt, factorial
```

If we simply do "import math", then math.sqrt(16) and math.factorial() are required.

```
print(sqrt(16))
print(factorial(6))
from mymodule import fib
from mymodule import fib, fib2
```

12.7 Import all Names – From import * Statement

The * symbol used with the from import statement is used to import all the names from a module to a current namespace.

```
# importing sqrt() and factorial from the module math
from math import *
print(sqrt(16))
print(factorial(6))
```

12.8 Locating Modules

Whenever a module is imported in Python the interpreter looks for several locations. First, it will check for the built-in module; if not found, then it looks for a list of directories defined in the sys.path. The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc. Python interpreter searches for the module in the following manner –

-First, it searches for the module in the current directory.

-If the module isn't found in the current directory, Python then searches each directory in the shell variable PYTHONPATH. The PYTHONPATH is an environment variable, consisting of a list of directories.

-If that also fails python checks the installation-dependent list of directories configured at the time Python is installed.

```
# importing sys module
import sys
# importing sys.path
print(sys.path)
```

12.9 Importing and renaming module

We can rename the module while importing it using the as keyword. For example, importing sqrt() and factorial from the module math

```
import math as gfg
print(gfg.sqrt(16))
print(gfg.factorial(6))
```

12.10 Pandas

Pandas is a powerful data manipulation and analysis library in Python. It introduces two primary data structures: Series and DataFrame.

Series are one-dimensional labeled array, akin to a column in a spreadsheet. See the example below:

```
import pandas as pd
# Creating a Pandas Series
my_series = pd.Series([10, 20, 30, 40, 50], name="My Numbers")
```

DataFrame is a two-dimensional table with rows and columns, similar to a spreadsheet or SQL table. See the example below:

```
# Creating a Pandas DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['Wonderland', 'Cityville', 'Metropolis']}
my_dataframe = pd.DataFrame(data)
```

Accessing and Manipulating DataFrames: Use methods to access and manipulate data within DataFrames.

```
ages = my_dataframe['Age']
```

```
filtered_data = my_dataframe[my_dataframe['Age'] > 30]
```

Importing and Exporting Data: Pandas supports reading and writing data from/to various formats, such as CSV, Excel, SQL, and more.

```
my_dataframe.to_csv('file.csv', index=False)
imported_data = pd.read_csv('file.csv')
```

Understanding NumPy and Pandas is fundamental for data manipulation and analysis in Python. NumPy provides a foundation for numerical operations, while Pandas simplifies data handling and offers powerful tools for working with structured data, making them indispensable in various aspects of biological data analysis and beyond.

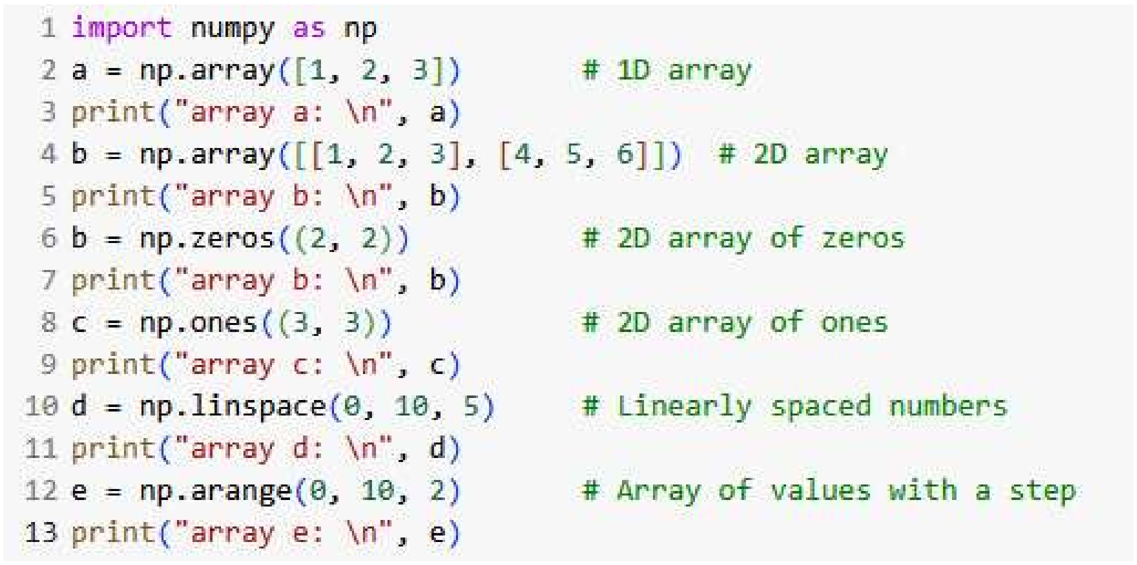
12.11 NumPy (Numerical Python)

NumPy is a Python library used for numerical computations. It provides high-performance multi-dimensional arrays and tools for working with them. Some of the key features are:

- **N-Dimensional Arrays**: `numpy.ndarray` is a powerful object for storing and manipulating large datasets.
- **Mathematical Functions**: Extensive functions for linear algebra, statistics, and Fourier transforms.
- **Broadcasting**: Perform operations on arrays of different shapes efficiently.
- **Integration with C/C++**: Works seamlessly with low-level programming languages for optimized computations.

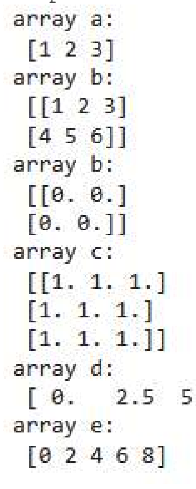
Some of the commonly used functions and operations of Numpy are:

a) **Creating Arrays**:



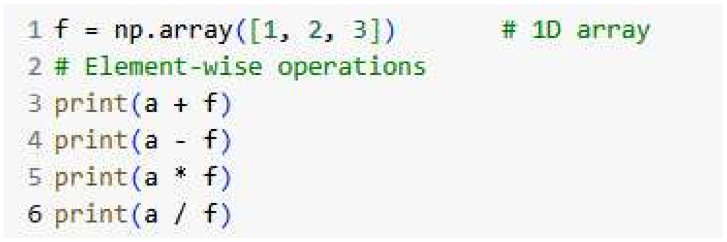
```
import numpy as np
a = np.array([1, 2, 3])         # 1D array
print("array a: \n", a)
b = np.array([[1, 2, 3], [4, 5, 6]])  # 2D array
print("array b: \n", b)
b = np.zeros((2, 2))            # 2D array of zeros
print("array b: \n", b)
c = np.ones((3, 3))             # 2D array of ones
print("array c: \n", c)
d = np.linspace(0, 10, 5)       # Linearly spaced numbers
print("array d: \n", d)
e = np.arange(0, 10, 2)         # Array of values with a step
print("array e: \n", e)
```

Output:



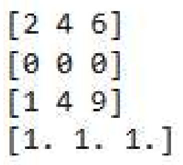
```
array a:
 [1 2 3]
array b:
 [[1 2 3]
 [4 5 6]]
array b:
 [[0. 0.]
 [0. 0.]]
array c:
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
array d:
 [ 0.   2.5  5.   7.5 10. ]
array e:
 [0 2 4 6 8]
```

b) **Array Operations**:



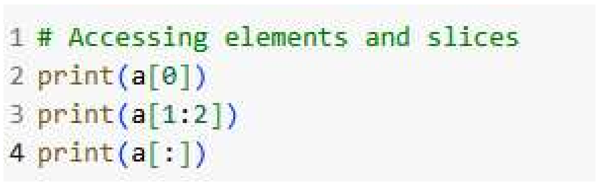
```python
f = np.array([1, 2, 3])       # 1D array
# Element-wise operations
print(a + f)
print(a - f)
print(a * f)
print(a / f)
```

Output:



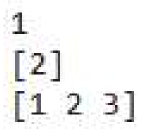
```
[2 4 6]
[0 0 0]
[1 4 9]
[1. 1. 1.]
```

c) **Indexing and Slicing**:



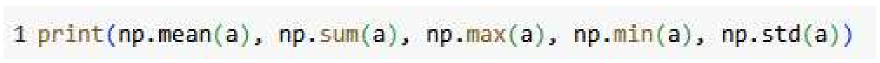
```python
# Accessing elements and slices
print(a[0])
print(a[1:2])
print(a[:])
```

Output:



```
1
[2]
[1 2 3]
```

d) **Mathematical Functions**:



```python
print(np.mean(a), np.sum(a), np.max(a), np.min(a), np.std(a))
```

Output:
```
2.0 6 3 1 0.816496580927726
```

e) **Reshaping Arrays**:

```python
# Reshape an array
g = np.array([1, 2, 3, 4])       # 1D array
h = g.reshape(2, 2),
print(h)
# Create a 2D NumPy array
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
print("Original 2D Array:")
print(array_2d)
# Flatten the array using flatten()
flattened_array = array_2d.flatten()
print("\nFlattened Array using flatten():")
print(flattened_array)
```

Output:
```
(array([[1, 2],
       [3, 4]]),)
Original 2D Array:
[[1 2 3]
 [4 5 6]]

Flattened Array using flatten():
[1 2 3 4 5 6]
```

12.12 Pandas

Pandas is a data manipulation and analysis library built on NumPy. It provides data structures like `Series` and `DataFrame` to handle structured data efficiently. Some of the key features are:

- **DataFrame and Series**: Core data structures for labeled and relational data.
- **Data Cleaning**: Tools for handling missing data, filtering, and transforming datasets.
- **Data Analysis**: Built-in functions for group operations, merging, and pivoting.
- **I/O Operations**: Read/write support for various formats like CSV, Excel, SQL, and JSON.

Commonly used functions and operations are:

a) **Creating Data Structures**:

```python
import pandas as pd
s = pd.Series([1, 2, 3, 4])           # 1D labeled array
df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})  # Tabular data
```

b) **Reading/Writing Data**:

```
df1 = pd.read_csv('file.csv')           # Read a CSV file
df1.to_excel('file.xlsx')               # Write to an Excel file
```

c) **DataFrame Operations**:

```
df.head(), df.tail()                    # View top/bottom rows
df.describe(), df.info()                # Summary statistics and structure
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2 entries, 0 to 1
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   A       2 non-null      int64
 1   B       2 non-null      int64
dtypes: int64(2)
memory usage: 160.0 bytes
(              A         B
 count  2.000000  2.000000
 mean   1.500000  3.500000
 std    0.707107  0.707107
 min    1.000000  3.000000
 25%    1.250000  3.250000
 50%    1.500000  3.500000
 75%    1.750000  3.750000
 max    2.000000  4.000000,
 None)
```

d) **Indexing and Selection**:

```
df['A'], df.iloc[0], df.loc[0, 'A']     # Select columns and rows
```

Output:

```
(0    1
 1    2
 Name: A, dtype: int64,
 A    1
 B    3
 Name: 0, dtype: int64,
 1)
```

e) **Data Cleaning**:

```
df.dropna(), df.fillna(0)               # Handle missing data
df.rename(columns={'A': 'X'})           # Rename columns
```

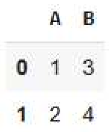
Output:

```
   X  B
0  1  3
1  2  4
```

f) **Data Aggregation**:

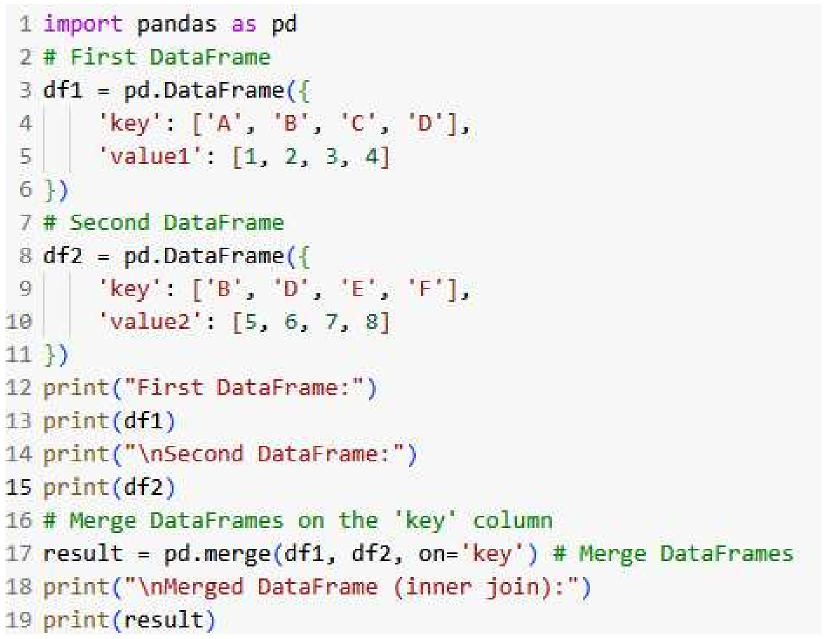
```
df.groupby('A').sum()                   # Group and aggregate
df.sort_values(by='A')                  # Sort by column
```

Output:



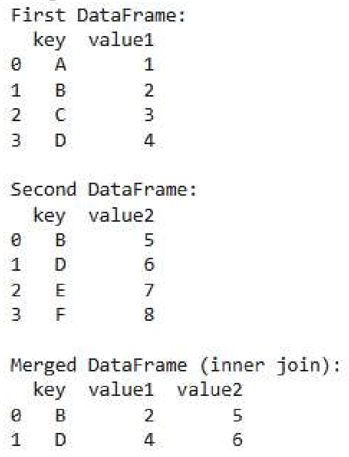
	A	B
0	1	3
1	2	4

g) **Merging and Joining**:



```python
import pandas as pd
# First DataFrame
df1 = pd.DataFrame({
    'key': ['A', 'B', 'C', 'D'],
    'value1': [1, 2, 3, 4]
})
# Second DataFrame
df2 = pd.DataFrame({
    'key': ['B', 'D', 'E', 'F'],
    'value2': [5, 6, 7, 8]
})
print("First DataFrame:")
print(df1)
print("\nSecond DataFrame:")
print(df2)
# Merge DataFrames on the 'key' column
result = pd.merge(df1, df2, on='key') # Merge DataFrames
print("\nMerged DataFrame (inner join):")
print(result)
```

Output:



```
First DataFrame:
  key  value1
0   A       1
1   B       2
2   C       3
3   D       4

Second DataFrame:
  key  value2
0   B       5
1   D       6
2   E       7
3   F       8

Merged DataFrame (inner join):
  key  value1  value2
0   B       2       5
1   D       4       6
```

13 Python Data Visualization

Data visualization is a critical aspect of data analysis, providing a visual representation of data patterns, trends, and insights. In Python, libraries like Matplotlib offer powerful tools for creating diverse and informative visualizations.

13.1 Matplotlib

Matplotlib is a comprehensive library for creating static, interactive, and animated visualizations in Python. Matplotlib is a versatile library for creating a wide range of plots, suitable for both beginners and advanced users. While its syntax can sometimes feel verbose, its power and flexibility make it indispensable for static data visualization tasks in Python. It is highly customizable and works well with NumPy and Pandas for data visualization. Some common applications are:

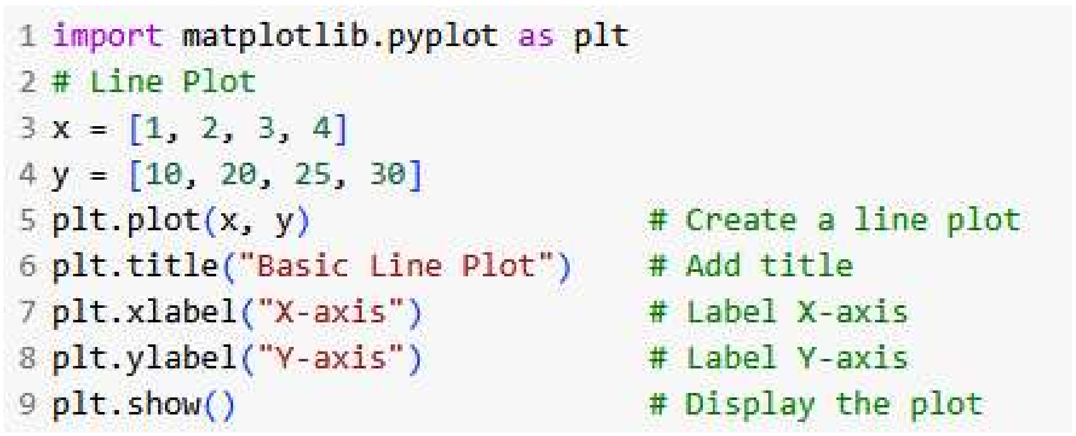
- Data Exploration: Visualize datasets for analysis.
- Data Presentation: Create professional-grade charts and graphs.
- Scientific Research: Plot experimental results, trends, and statistical data.

The key features of this module are:

- **2D Plotting**: Supports line plots, scatter plots, bar plots, histograms, and more.
- **Customization**: Extensive control over plot elements like labels, colors, markers, and annotations.
- **Interactive Plots**: Works seamlessly with Jupyter notebooks and supports interactive backends.
- **Integration**: Compatible with other Python libraries like NumPy, Pandas, and Seaborn.

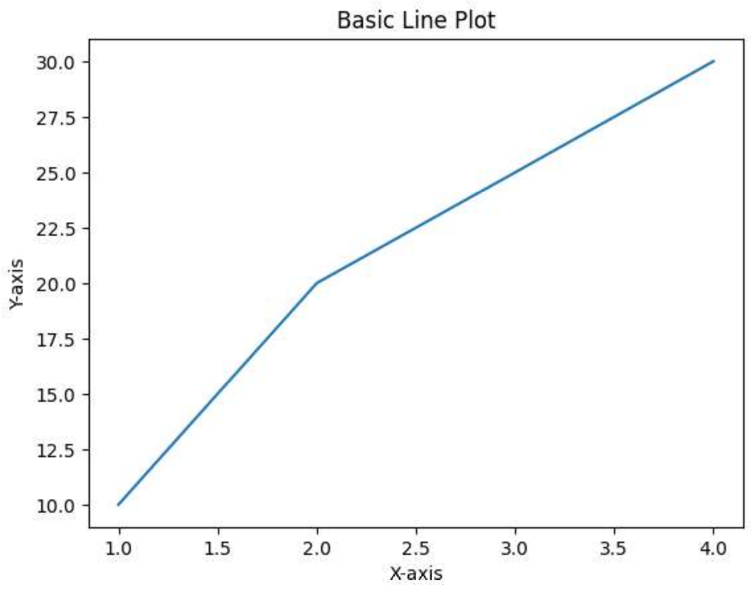
The commonly used functions and operations are:

- **Basic Plotting**: Matplotlib's `pyplot` module allows for simple line plotting



```python
import matplotlib.pyplot as plt
# Line Plot
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]
plt.plot(x, y)                  # Create a line plot
plt.title("Basic Line Plot")    # Add title
plt.xlabel("X-axis")            # Label X-axis
plt.ylabel("Y-axis")            # Label Y-axis
plt.show()                      # Display the plot
```

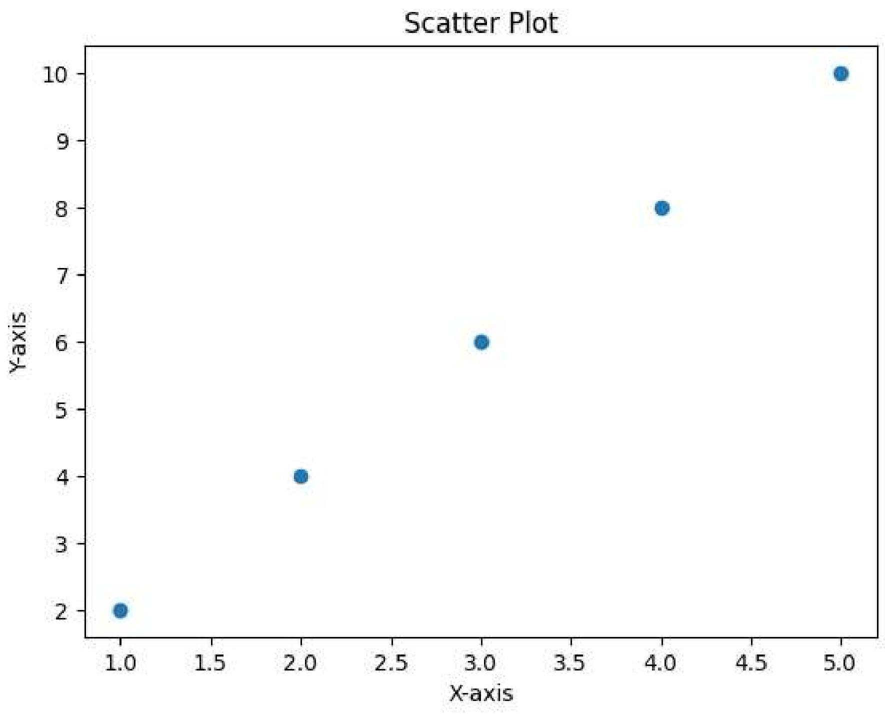
`The code will give the following output:`



- **Scatter Plot:** Scatter plots are useful for visualizing relationships between two variables.

```python
# Scatter plot
plt.scatter(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot')
plt.show()
```

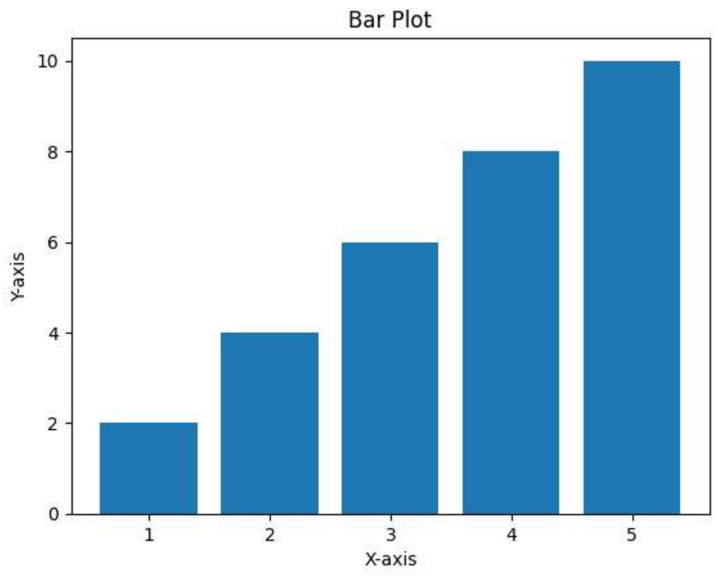
Output:



- **Bar Plot:** Bar plots are suitable for comparing categories.

```
# Bar plot
plt.bar(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Bar Plot')
plt.show()
```

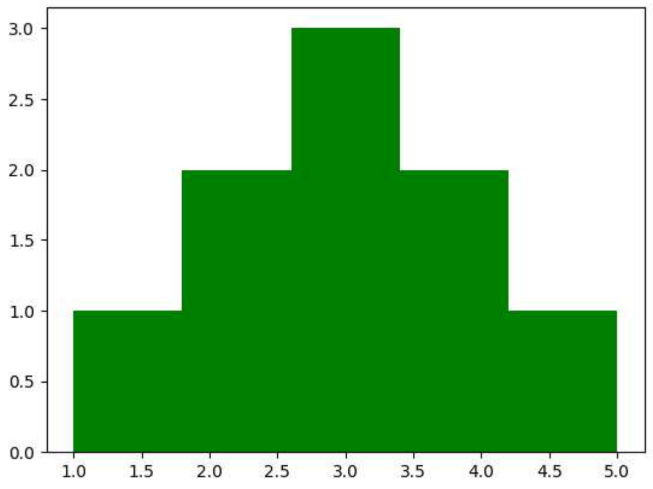
```
Output:
```



- **Historgram**

```
# Histogram
data = [1, 2, 2, 3, 3, 3, 4, 4, 5]
plt.hist(data, bins=5, color='green')
```

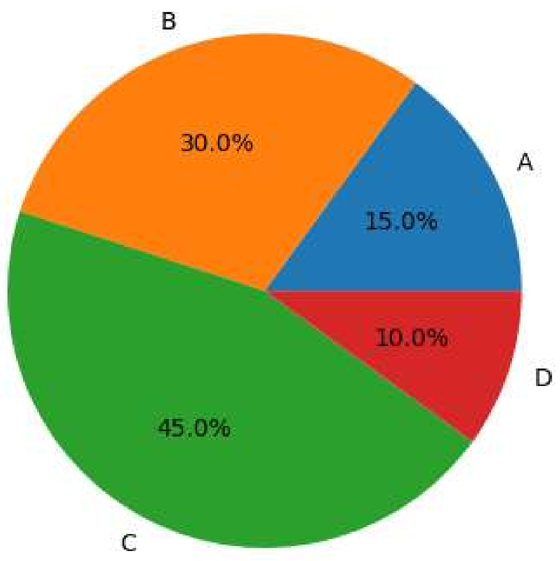
```
Output:
```



- **Pie chart**

```
# Pie Chart
labels = ['A', 'B', 'C', 'D']
sizes = [15, 30, 45, 10]
plt.pie(sizes, labels=labels, autopct='%1.1f%%')
```

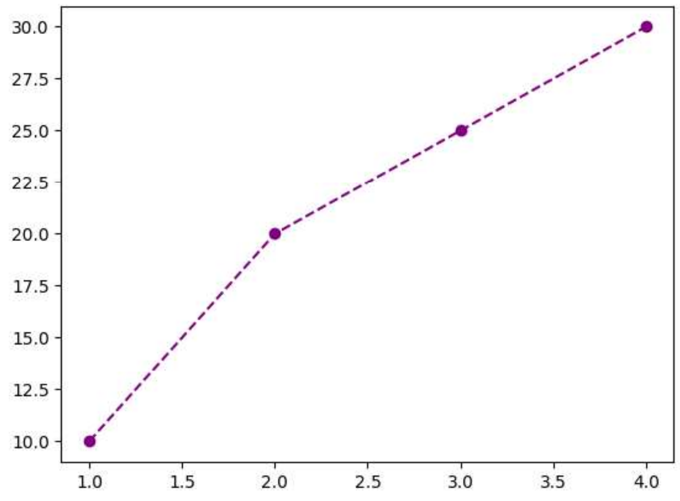
Output:



- Customizing Plots

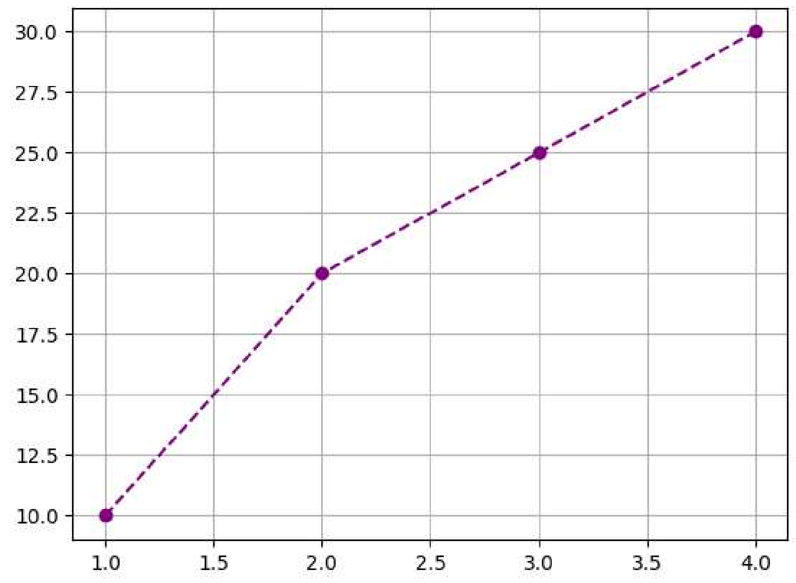
```
# Line style, markers, and colors
plt.plot(x, y, linestyle='--', marker='o', color='purple')
```

Output:



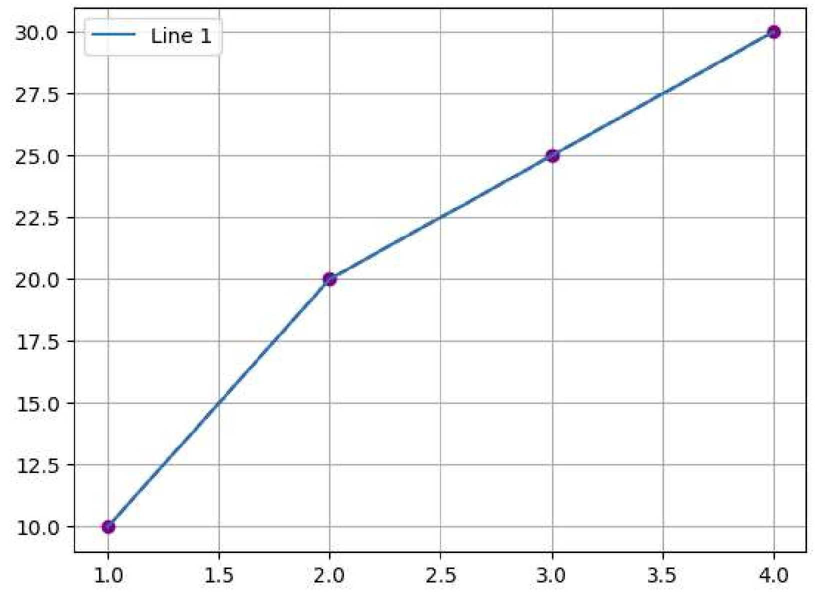
```
# Adding a grid
plt.plot(x, y, linestyle='--', marker='o', color='purple')
plt.grid(True)
```

Output:



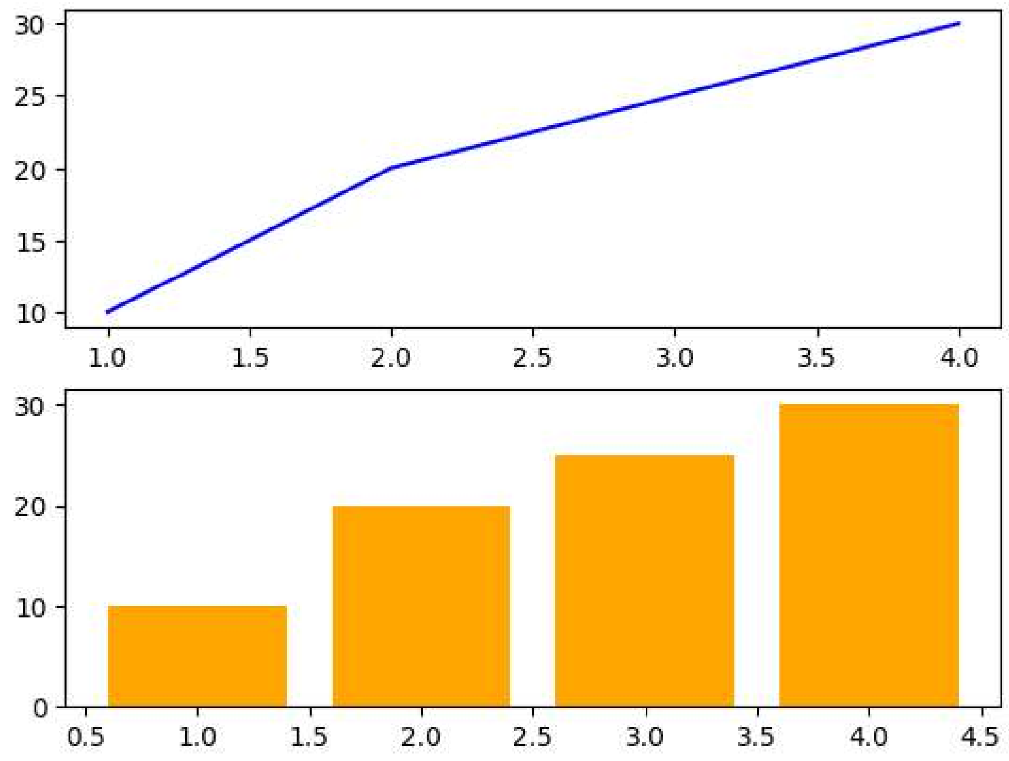
```
# Legends
plt.plot(x, y, linestyle='--', marker='o', color='purple')
plt.grid(True)
plt.plot(x, y, label="Line 1")
plt.legend()
```

Output:



```python
# Subplots
plt.subplot(2, 1, 1)           # (rows, columns, plot number)
plt.plot(x, y, color='blue')
plt.subplot(2, 1, 2)
plt.bar(x, y, color='orange')
```

Output:



14 Using Python in Biological Research

With advanced learning of Python, many problems of data analysis in the Biological Sciences can be addressed using Python. For example, analyzing large-scale genomic data to identify genes, mutations, and regulatory elements, Identifying potential drug candidates and understanding their interactions with biological targets, classifying diseases based on patient data, such as gene expression profiles or clinical features, predicting the three-dimensional structure of proteins from amino acid sequences, extracting information from a vast amount of scientific literature, studying genetic variations within populations. Python's versatility and extensive ecosystem of libraries empower researchers and practitioners to address diverse biological challenges. It can help in automating repetitive tasks, integrating tools and pipelines, web scraping for data collection, performing statistical analysis and hypothesis testing, predictive modeling using machine learning, data cleaning, and pre-processing. Python's adaptability and extensive community support make it an invaluable tool for addressing a wide range of biological problems. Its ease of use, rich libraries, and ecosystem make Python an essential programming language in the field of biological research and data analysis.