

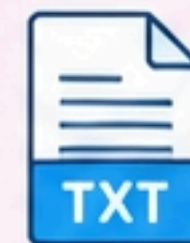
Types of Files

Files can be mainly classified into two types based on the data they store and how they are handled in Python.

1. Text Files

- Store data as human-readable text.
- Examples: .txt, .csv, .log, .json, .html
- Opened in text mode (default).

The data is stored as characters (strings).



2. Binary Files

- Store data in binary (non-text) format.
- Examples: .jpg, .png, .mp3, .exe, .dat
- Opened in binary mode ("rb", "wb").

The data is stored as bytes (numbers).



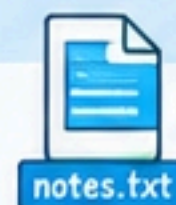
Working with Text Files

Text files are opened in text mode ("r", "w", "a").
Python automatically handles the **encoding** (default: utf-8).

1 Create a Text File

Use mode "w" (write). It creates a new file or overwrites an existing file.

```
1 # create a text file and write data
2 with open("notes.txt", "w") as f:
3     f.write("Hello, Python!\nThis is a text file.")
```



2 Read a Text File

Use mode "r" (read).

```
1 # read data from a text file
2 with open("notes.txt", "r") as f:
3     content = f.read()
4     print(content)
```

Output:

```
Hello, Python!
This is a text file.
```

3 Write / Append to a Text File

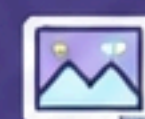
Use mode "a" (append) to add data without deleting existing content.

```
1 # append data to a text file
2 with open("notes.txt", "a") as f:
3     f.write("\nThis line is added.")
```



Modes for text files:

```
"r" – read
"w" – write (overwrite)
"a" – append
"r+" – read & write
```



Working with Binary Files

Binary files are opened in binary mode ("rb", "wb", "ab").
They are read and written as bytes (not text).

1 Create a Binary File

Use mode "wb" (write binary).

```
1 # write binary data (e.g., an image file)
2 with open("image.jpg", "wb") as f:
3     f.write(b"\xFF\xD8\xFF\xE0...") # binary data
```



2 Read a Binary File

Use mode "rb" (read binary).

```
1 # read binary data from a file
2 with open("image.jpg", "rb") as f:
3     data = f.read()
4     print(type(data)) # <class 'bytes'>
```

Output:

```
<class 'bytes'>

The data is returned
as a bytes object.
```

3 Write / Append to a Binary File

Use mode "ab" (append binary) to add data.

```
1 # append binary data to a file
2 with open("data.bin", "ab") as f:
3     f.write(b"\x01\x02\x03") # bytes data
```



Modes for binary files:

```
"rb" – read
"wb" – write (overwrite)
"ab" – append
"r+b" – read & write
```



Key Takeaways

- ✓ Text files store human-readable data (strings).
- ✓ Binary files store data in bytes (non-text).

- ✓ Use the right file mode: "r", "w", "a" for text files and "rb", "wb", "ab" for binary files.
- ✓ Always use the with statement to automatically close the file.



The Pickle Module & Reading and Writing CSV Files in Python

Save Python objects • Work with CSV data • Make your programs more powerful

Serialize
• Deserialize
Work with Data



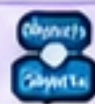
The Pickle Module

The pickle module in Python is used to serialize and deserialize Python objects. It converts Python objects into a byte stream (and back), so they can be saved to a file or transferred over a network.



Key Features

- Works with most Python objects (lists, dicts, tuples, etc.).
- Preserves the structure and data of the object.
- Useful for saving data, caching, and inter-process communication.
- Not secure against untrusted data (do not load pickled data from unknown sources).



Common Methods

Method	Description
<code>pickle.dump(obj, file)</code>	Serialize (write) an object to a file.
<code>pickle.load(file)</code>	Deserialize (read) an object from a file.
<code>pickle.dumps(obj)</code>	Serialize to bytes (in memory).
<code>pickle.loads(bytes)</code>	Deserialize from bytes (in memory).



Example: Using Pickle

1 Writing (Pickling) an Object

```
import pickle

# Create a Python object
data = {'name': 'Alice', 'age': 25,
        'skills': ['Python', 'C++', 'DSA']}

# Write to a file using pickle
with open('data.pkl', 'wb') as f:
    pickle.dump(data, f)

print("Object pickled successfully!")
```

The object is saved in binary format in data.pkl.

2 Reading (Unpickling) an Object

```
import pickle

# Read the object from file
with open('data.pkl', 'rb') as f:
    loaded_data = pickle.load(f)

print("Loaded data:", loaded_data)
```

The original Python object is reconstructed.



Use Cases

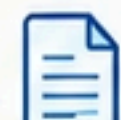
- Saving machine learning models.
- Caching data for faster access.
- Storing complex objects (e.g., lists, dictionaries, custom classes).



Python
Object



pickle
(serialize)



.pkl file



pickle
(deserialize)



Python
Object



Reading and Writing CSV Files in Python

CSV (Comma Separated Values) is a simple file format used to store tabular data. Python's built-in csv module provides easy ways to read and write CSV files.



Useful Functions in csv Module

Function	Description
<code>csv.reader()</code>	Reads data from a CSV file (returns rows).
<code>csv.writer()</code>	Writes data to a CSV file.
<code>csv.DictReader()</code>	Reads CSV file as dictionaries (with headers).
<code>csv.DictWriter()</code>	Writes dictionaries to a CSV file (with headers).



Example 1: Reading a CSV File

```
import csv

# Read and print data from a CSV file
with open('students.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

students.csv
(content)

```
students.csv
Name,Age,Course
Alice,20,Python
Bob,22,Data Science
Charlie,21,Web Development
```

Output:

```
['Name', 'Age', 'Course']
['Alice', '20', 'Python']
['Bob', '22', 'Data Science']
['Charlie', '21', 'Web Development']
```



Example 2: Writing a CSV File

```
import csv

# Write data to a CSV file
data = [['Name', 'Age', 'Course'],
        ['Alice', 20, 'Python'],
        ['Bob', 22, 'Data Science'],
        ['Charlie', 21, 'Web Development']]

with open('output.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data)
```

output.csv
(content)

output.csv

```
Name,Age,Course
Alice,20,Python
Bob,22,Data Science
Charlie,21,Web Development
```



The `newline=` argument is important on Windows to avoid blank lines between rows.



Quick Recap

✓ Pickle module → serialize/deserialize Python objects (works with most data types).

✓ CSV module → read and write tabular data (using reader, writer, DictReader, DictWriter).

✓ Use pickle for complex objects and caching.
✓ Use CSV for structured, human-readable data.

Save
Data
Work
Smarter!



Reading and Writing of CSV and JSON Files in Python

CSV = Tabular data | JSON = Structured data | Python makes it easy!

Read · Write
· Handle Data
in Python



What is CSV?

- CSV (Comma Separated Values) stores data in a **tabular** format (rows and columns).
- Each line is a row, values are separated by **commas**.
- Commonly used for spreadsheets and data exchange.

Example CSV

Name	Age	Course
Alice	20	Python
Bob	22	Data Science
Charlie	21	Web Dev

(saved as students.csv)



What is JSON?

- JSON (JavaScript Object Notation) stores data in key-value pairs (like a dictionary).
- Supports nested data (lists, objects).
- Widely used in web APIs and modern applications.

Example JSON

```
[
  { "name": "Alice", "age": 20,
    "course": "Python" },
  { "name": "Bob", "age": 22,
    "course": "Data Science" },
  { "name": "Charlie", "age": 21,
    "course": "Web Dev" }
]
```

(saved as students.json)



Working with CSV Files

1 Reading a CSV File

Use the csv module's reader() to read rows from a CSV file.

```
import csv
with open('students.csv', 'r', newline='', encoding='utf-8') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Output:

```
['Name', 'Age', 'Course']
['Alice', '20', 'Python']
['Bob', '22', 'Data Science']
['Charlie', '21', 'Web Dev']
```

2 Writing to a CSV File

Use the csv module's writer() to write data to a CSV file.

```
import csv
data = [
    ['Name', 'Age', 'Course'],
    ['Alice', 20, 'Python'],
    ['Bob', 22, 'Data Science'],
    ['Charlie', 21, 'Web Dev']
]
with open('students.csv', 'w', newline='', encoding='utf-8') as file:
    writer = csv.writer(file)
    writer.writerows(data)
```

This creates a file
students.csv with
the given data.



students.csv



Useful CSV Functions

- ✓ csv.reader() – read rows
- ✓ csv.writer() – write rows
- ✓ DictReader() / DictWriter() – work with dictionaries (column headers)



Working with JSON Files

1 Reading a JSON File

Use the json module's load() to read JSON data from a file.

```
import json
with open('students.json', 'r', encoding='utf-8') as file:
    data = json.load(file)
    print(data)
```

Output:

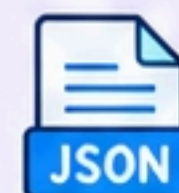
```
[{'name': 'Alice', 'age': 20,
  'course': 'Python'},
 {'name': 'Bob', 'age': 22,
  'course': 'Data Science'},
 {'name': 'Charlie', 'age': 21,
  'course': 'Web Dev'}]
```

2 Writing to a JSON File

Use the json module's dump() to write data to a JSON file.

```
import json
data = [
    {"name": "Alice", "age": 20, "course": "Python"},
    {"name": "Bob", "age": 22, "course": "Data Science"},
    {"name": "Charlie", "age": 21, "course": "Web Dev"}
]
with open('students.json', 'w', encoding='utf-8') as file:
    json.dump(data, file, indent=4)
```

This creates a file
students.json with
formatted JSON data.



students.json



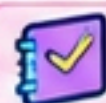
Useful JSON Functions

- ✓ json.load() – read JSON file
- ✓ json.dump() – write JSON file
- ✓ json.loads() / json.dumps() – work with JSON strings



Key Points to Remember

- ✓ Use the csv module for CSV files.
- ✓ Use the json module for JSON files.
- ✓ Always open files with the correct mode:
 - 'r' (read), 'w' (write), 'a' (append).



File Modes

- 'r' – read (default, file must exist)
- 'w' – write (overwrites or creates)
- 'a' – append (adds to end)
- 'r+' – read and write (file must exist)
- 'w+' – read and write (overwrites or creates)



Pro Tips

- ✓ Use newline="" when working with CSV files (in Windows).
- ✓ Use indent=4 for pretty JSON output.
- ✓ Handle exceptions (try-except) when reading files.
- ✓ Use with statement for automatic file closing.

Better
Code
Better
Data!



Exception Handling

Try - Except - Else - Finally Block in Python

Handle errors gracefully • Keep your program running • Be more robust!

Catch Errors
Handle them
Continue!

1. What is Exception Handling?

Exception handling is a way to deal with runtime errors (exceptions) in Python. It lets you handle the error gracefully instead of crashing the program.

It helps your program stay safe,
even when something goes wrong!

Error
handled!

2. The try-except-else-finally Block

try : Code that may raise an exception.

except : Code to handle the exception.

else : Code that runs if no exception occurs.

finally : Code that always runs (optional).

3. How It Works?

- 1 Run the code inside **try**.
- 2 If an error occurs, control moves to the matching **except** block.
- 3 If no error occurs, the **else** block runs (optional).
- 4 The **finally** block runs always, no matter what (optional).

4. Example: Division Program

This program takes two numbers and performs division. It handles the case when the second number is 0.

```
# exception handling with try-except-else-finally
try:
    num1 = float(input("Enter first number: "))
    num2 = float(input("Enter second number: "))
    result = num1 / num2
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
else:
    print("Division completed successfully.")
finally:
    print("This block always runs (cleanup code).")
```

Code that may
raise an exception
(e.g., ZeroDivisionError).

Handles the specific
exception.

Runs if no exception
occurs.

Always runs
(used for cleanup,
closing files, etc.).

5. Output Scenarios

Case 1: Valid Input (no error)

Input:

Enter first number: 10
Enter second number: 2

Output:

Result: 5.0
Division completed successfully.
This block always runs
(cleanup code).

✓ No error → **else** block runs.

Case 2: Division by Zero

Input:

Enter first number: 10
Enter second number: 0

Output:

Error: Division by zero
is not allowed.
This block always runs
(cleanup code).

✓ Error → **except** block runs.

6. Key Points to Remember

- ✓ Use **try-except** to handle errors.
- ✓ Use **else** for code that runs when no error occurs.
- ✓ Use **finally** for cleanup (always runs).
- ✓ You can handle specific exceptions (e.g., ZeroDivisionError, ValueError).
- ✓ Always keep your code inside **try** block as small as possible.

7. Common Exceptions in Python

Exception	Meaning
ZeroDivisionError	Division by zero
ValueError	Invalid value
TypeError	Wrong data type
IndexError	Index out of range
FileNotFoundError	File not found

8. Pro Tips

- ✓ Catch specific exceptions (not just Exception).
- ✓ Use the **finally** block to close files, release resources, or clean up.
- ✓ Log errors for debugging.
- ✓ Don't ignore exceptions silently.
- ✓ Keep your code clean and readable.

Handle
Errors
Like a Pro!

➤ Better Error Handling = Stronger, More Reliable Python Programs ➤



Raise Statement, Hierarchy of Exceptions and Adding Exceptions in Python

Raise an error • Understand exception hierarchy • Create your own exceptions

Handle Errors Like a Pro!

1. raise Statement

The raise statement is used to manually trigger an exception in Python. It helps you signal that something went wrong, even if Python does not raise it automatically.

Syntax

```
raise ExceptionType("Error message")
```

Where to use?

- ✓ When a condition is not met.
- ✓ To validate input or data.
- ✓ To handle errors in custom logic.

Example

```
def check_age(age):  
    if age < 18:  
        raise ValueError("Age must be 18 or above")  
    return "Valid age"  
  
try:  
    print(check_age(16))  
except ValueError as e:  
    print("Error:", e)
```

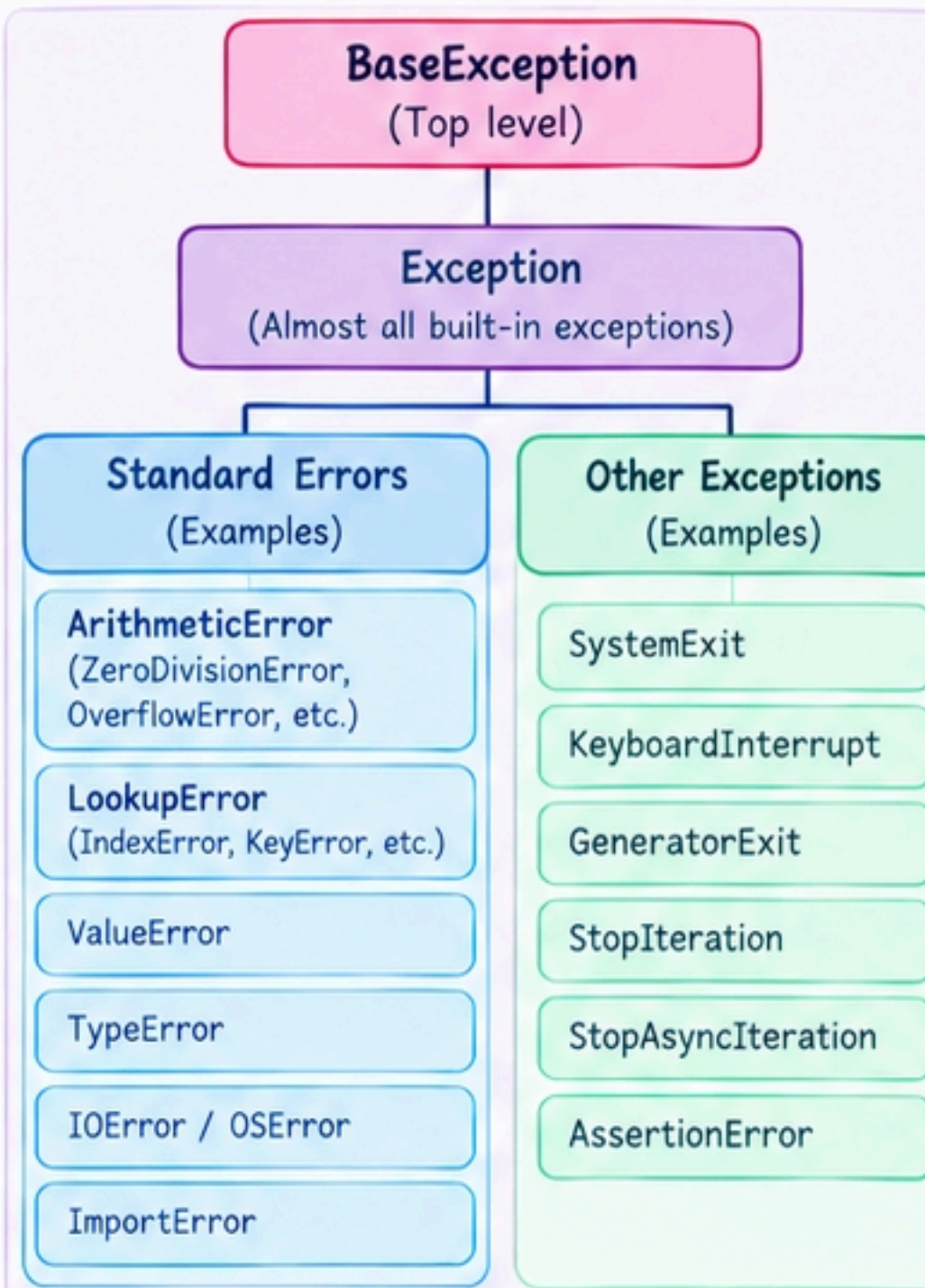


Output:
Error: Age must be 18 or above

raise helps you control the flow of your program.

2. Hierarchy of Exceptions

Python has a built-in hierarchy of exceptions. All exceptions are derived from the BaseException class. The most common exceptions are derived from Exception class.



You can catch specific exceptions or the base class Exception to handle multiple exceptions at once.

3. Adding Exceptions in Python

You can create your own custom exceptions by defining a class that inherits from the Exception class (or any of its child classes).

Syntax

```
class CustomError(Exception):  
    def __init__(self, message):  
        super().__init__(message)
```

Example: Custom Exception

```
class InvalidMarksError(Exception):  
    """Exception raised for invalid marks"""  
    pass  
  
# Using the custom exception  
try:  
    marks = int(input("Enter marks: "))  
    if marks < 0 or marks > 100:  
        raise InvalidMarksError("Marks must be between 0 and 100")  
    print("Valid marks:", marks)  
except InvalidMarksError as e:  
    print("Error:", e)
```



Output:
Error: Marks must be between 0 and 100

Benefits

- ✓ Makes your code more readable.
- ✓ Helps you handle specific error cases.
- ✓ Improves code organization and maintainability.



Key Takeaways



raise statement

- Manually triggers exceptions.
- Use it for validation and control.
- Example: raise ValueError("msg")



Exception hierarchy

- BaseException → Exception
- Many built-in exceptions
- Use specific or Exception to catch multiple.



Adding exceptions

- Inherit from Exception.
- Define a clear error message.
- Use raise to throw it.

Better Error Handling = More Robust Programs!

1. Histogram

A histogram is used to show the distribution of a continuous variable. It groups data into bins and shows the frequency (count) in each bin.

Key Points

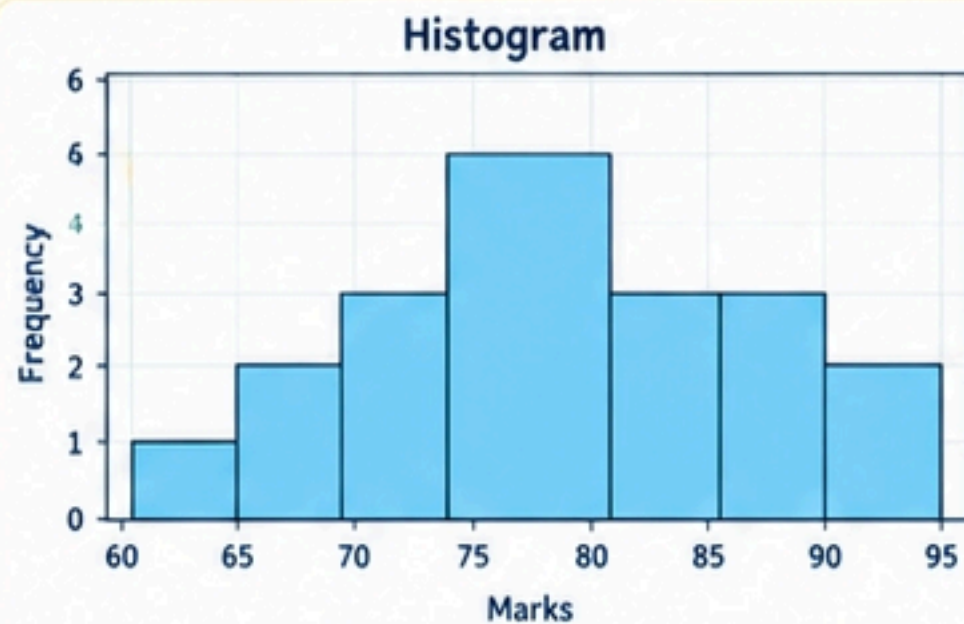
- Used for continuous data (e.g., marks, height, age).
- X-axis: range (bins), Y-axis: frequency.
- Shows the shape, spread and outliers of data.

Python Code

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

# Sample data
data = [72, 85, 67, 90, 78, 88, 95, 62, 70, 76,
        82, 69, 73, 91, 87, 65, 80, 77, 83, 68]

plt.hist(data, bins=5, color='skyblue', edgecolor='black')
plt.title('Histogram')
plt.xlabel('Marks')
plt.ylabel('Frequency')
plt.show()
```



What it tells us?

The histogram shows how many marks fall in each range. We can easily see where most values are concentrated.

Libraries Used

-  **matplotlib.pyplot** – for plotting graphs
-  **numpy** – for numerical operations (e.g., linspace, sin, cos)

2. Pie Charts

A pie chart is used to show how different categories contribute to a whole. Each slice represents a percentage (or proportion) of the total.

Key Points

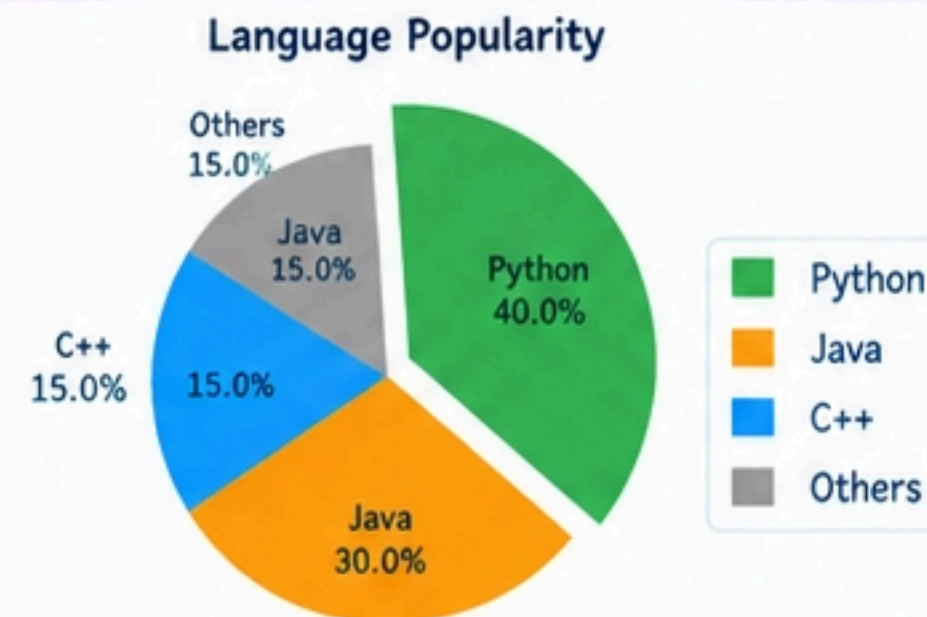
- Used for categorical data (e.g., subjects, sales).
- Shows part-to-whole relationship.
- Percentages add up to 100%.

Python Code

```
import matplotlib.pyplot as plt

categories = ['Python', 'Java', 'C++', 'Others']
values = [40, 30, 15, 15]
colors = ['#4CAF50', '#FF9800', '#2196F3', '#9E9E9E']

plt.pie(values, labels=categories, autopct='%1.1f%%',
        startangle=140, colors=colors, explode=[0,0.05,0,0,0])
plt.title('Language Popularity')
plt.axis('equal')
plt.show()
```



What it tells us?

Python is the most popular language (40%), followed by Java (30%), C++ (15%) and Others (15%).

Quick Tips

- ✓ Use `plt.figure()` to create multiple plots.
- ✓ Customize colors, labels, titles and legends.
- ✓ Use `grid()` for better readability.
- ✓ Adjust figure size with `figsize=(width, height)`.

3. Sine and Cosine Curves

Sine and cosine are trigonometric functions. They are periodic and are widely used in science, engineering, and data analysis.

Key Points

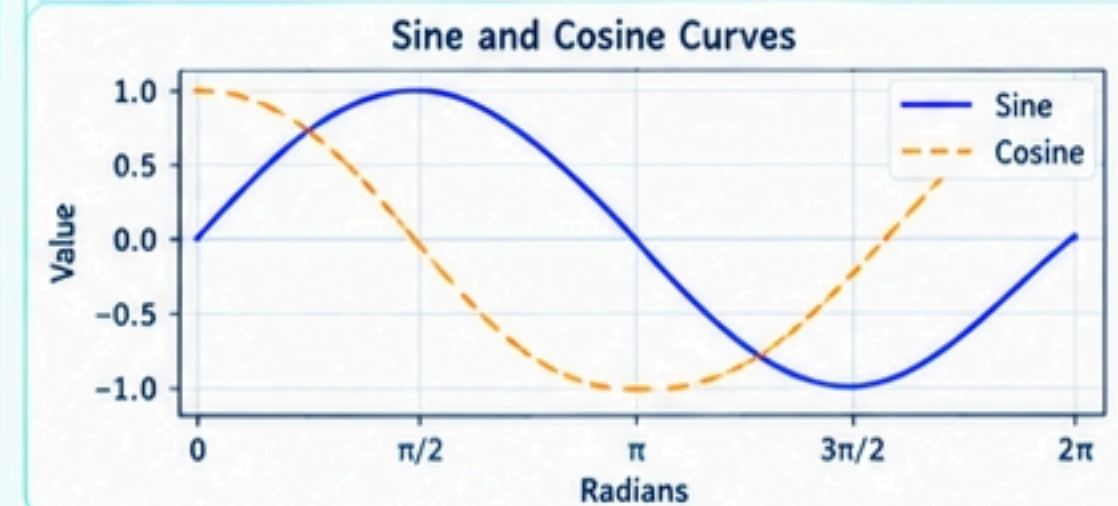
- Sine and cosine have the same period (2π).
- Sine starts at 0, cosine starts at 1.
- Used in waveforms, signal processing and modelling.

Python Code

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

# Create values from 0 to 2π
x = np.linspace(0, 2*np.pi, 100)
y_sin = np.sin(x)
y_cos = np.cos(x)

plt.plot(x, y_sin, label='Sine', color='blue', linewidth=2)
plt.plot(x, y_cos, label='Cosine', color='orange',
        linestyle='--', linewidth=2)
plt.title('Sine and Cosine Curves')
plt.xlabel('Radians')
plt.ylabel('Value')
plt.xticks([0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi],
        ['0', 'π/2', 'π', '3π/2', '2π'])
plt.legend()
plt.grid(True)
plt.show()
```



What it tells us?

The graph shows the wave-like pattern of sine and cosine functions. Sine starts at 0, while cosine starts at 1.

