**Advance Python Programming**

**Printing on Screen**

1. **Introduction to the print() function in Python.**

**ANS:** The print() function in Python is one of the most commonly used built-in functions. It allows you to output information to the console or standard output. This is essential for displaying messages, debugging code, and interacting with users.

Syntax**:**

print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Parameters**:**

\*objects: One or more objects to be printed. These objects are separated by commas. You can pass strings, numbers, lists, etc. The print() function will convert them to string representations automatically.

sep: A string inserted between the objects (default is a space).

end: A string appended at the end of the output (default is a newline \n).

file: The stream where the output is printed (default is sys.stdout, which is usually the console).

flush: A boolean that, when set to True, forces the flushing of the output buffer (default is False).

Examples**:**

1. Basic usage:

print("Hello, World!")

This will print:

Hello, World!

2. Printing multiple items:

print("The answer is", 42)

This will print:

The answer is 42

3. Custom separator:

print("apple", "banana", "cherry", sep=", ")

This will print:

apple, banana, cherry

4. Custom end character:

print("Hello", end=" ")

print("World!")

This will print:

Hello World!

5. Printing to a file:

with open("output.txt", "w") as f:

print("Hello, File!", file=f)

This will write "Hello, File!" to the output.txt file.

1. **Formatting outputs using f-strings and format().**

**ANS:** In Python, formatting outputs can be done using two popular methods: **f-strings** (formatted string literals) and the .format() method. Both allow you to create dynamic strings by embedding values or expressions within a string. Let's dive into both methods and explore how they work.

1. **F-strings (Formatted String Literals)**

Introduced in Python 3.6, **f-strings** are a simple and efficient way to embed expressions inside string literals. You prefix the string with the letter f or F, and then use curly braces {} to insert variables or expressions.

Basic Syntax:

f"string with {expression}"

Example:

name = "Alice"

age = 30

print(f"Hello, my name is {name} and I am {age} years old.")

**Output:**

Hello, my name is Alice and I am 30 years old.

Expressions Inside f-strings:

You can embed not just variables, but any expressions inside the curly braces.

x = 5

y = 10

print(f"The sum of {x} and {y} is {x + y}.")

**Output:**

The sum of 5 and 10 is 15.

Formatting with f-strings:

You can format numbers and other data types directly in f-strings, such as limiting the number of decimal places for floating-point numbers.

pi = 3.14159

print(f"Pi to two decimal places: {pi:.2f}")

**Output:**

Pi to two decimal places: 3.14

2. **The** .format() **Method**

Before f-strings were introduced, the .format() method was the go-to way of formatting strings in Python. It allows you to insert values into a string by referencing placeholders {} within the string.

Basic Syntax:

"string with {}".format(value)

Example:

name = "Bob"

age = 25

print("Hello, my name is {} and I am {} years old.".format(name, age))

**Output:**

Hello, my name is Bob and I am 25 years old.

Positional and Keyword Arguments:

You can refer to arguments by their position or name using .format().

# Positional arguments

print("Hello, my name is {} and I am {} years old.".format(name, age))

# Using index to refer to the values

print("Hello, my name is {0} and I am {1} years old.".format(name, age))

# Keyword arguments

print("Hellon, my name is {name} and I am {age} years old.".format(name=name, age=age))

Formatting with .format():

You can also format numbers and other values using .format().

pi = 3.14159

print("Pi to two decimal places: {:.2f}".format(pi))

**Output:**

Pi to two decimal places: 3.14

**Comparison: f-strings vs.** .format()

Here’s a quick comparison of the two methods for formatting strings:

**Readability**:

**f-strings**: More readable and concise, especially when embedding variables or expressions.

**.format()**: Requires additional syntax (e.g., 0, 1, or name inside curly braces).

**Performance**:

**f-strings** are generally faster because they are evaluated at runtime.

**.format()** is a bit slower because it processes the format specifiers at runtime.

**Flexibility**:

**f-strings** are generally more flexible and can handle expressions directly within the braces.

**.format()** offers a more complex system with positional and keyword arguments, useful in some scenarios for formatting.

**Examples:**

1. **Positional Arguments**

**Using f-string:**

name = "John"

age = 40

print(f"My name is {name} and I am {age} years old.")

**Using .format() method:**

print("My name is {} and I am {} years old.".format(name, age))

Both will output:

My name is John and I am 40 years old.

2. **Formatting Numbers**

**Using f-string:**

pi = 3.14159

print(f"Pi to 2 decimal places: {pi:.2f}")

**Using .format() method:**

print("Pi to 2 decimal places: {:.2f}".format(pi))

Both will output:

Pi to 2 decimal places: 3.14

3. **Aligning Text**

**Using f-string:**

print(f"{'apple':<10}{'banana':^10}{'cherry':>10}")

**Using .format() method:**

print("{:<10}{:^10}{:>10}".format('apple', 'banana', 'cherry'))

Both will output:

apple banana cherry

**Reading Data** **from Keyboard**

1. **Using the input() function to read user input from the keyboard.**

**ANS:** The input() function in Python allows you to read data entered by the user via the keyboard. It pauses the program, waits for the user to type something, and then returns that input as a string. This is very useful for interactive programs that need to get information from the user.

Syntax**:**

input(prompt)

**prompt (optional)**: A string that is displayed to the user, prompting them to enter some input. If no prompt is given, Python just waits for the input without showing any message.

**Return Value:**

The input() function always returns the data as a **string**. Even if the user enters a number, it will be treated as a string unless explicitly converted.

**Examples:**

**1. Basic Usage of input()**

name = input("What is your name? ")

print(f"Hello, {name}!")

**Output (example interaction):**

What is your name? Alice

Hello, Alice!

Here, the program prompts the user to input their name, and then prints a greeting using the name they entered.

**2. Converting User Input to Another Data Type**

Since input() returns data as a string, you may want to convert it to other types like int or float. Here’s how you can handle numeric input:

age = input("How old are you? ")

age = int(age) # Convert the input to an integer

print(f"You are {age} years old.")

**Output (example interaction):**

How old are you? 25

You are 25 years old.

In this case, the program takes the input string and converts it to an integer using int().

**3. Handling Floating Point Numbers**

If you need to handle floating point numbers, use float() to convert the string input into a float:

price = input("Enter the price of the item: ")

price = float(price) # Convert to a floating-point number

print(f"The price is ${price:.2f}")

**Output (example interaction):**

Enter the price of the item: 19.99

The price is $19.99

**4. Handling Multiple Inputs on One Line**

You can also read multiple pieces of data at once by splitting the input string. Here’s an example where the user enters their name and age separated by a comma:

name, age = input("Enter your name and age separated by a comma: ").split(",")

age = int(age) # Convert age to an integer

print(f"Name: {name}, Age: {age}")

**Output (example interaction):**

Enter your name and age separated by a comma: John, 30

Name: John, Age: 30

In this example, the input is split by the comma using .split(","), and then the age is converted to an integer.

**5. Error Handling for Invalid Input**

Sometimes, users might enter data that can't be converted into the expected type (like entering "abc" when a number is expected). You can handle such cases using try-except blocks:

try:

age = int(input("How old are you? "))

print(f"You are {age} years old.")

except ValueError:

print("Invalid input! Please enter a valid number.")

**Output (example interaction):**

How old are you? abc

Invalid input! Please enter a valid number.

**Key Points to Remember:**

**The input() function always returns a string**, even if the user enters numbers. You need to convert it to the desired type (e.g., int(), float()).

You can use a **prompt** to ask the user for input, making the program more interactive.

To handle errors (like non-numeric input when expecting a number), you can use **exception handling** with try-except.

Using input() is a simple and powerful way to get user input in your programs, and by converting and handling the input correctly, you can create interactive and robust applications.

1. **Converting user input into different data types (e.g., int, float, etc.).**

**ANS:** In Python, the input() function always returns the user's input as a **string**. If you want to work with numerical data, such as integers or floating-point numbers, you need to explicitly **convert** the input string to the desired data type.

Here’s how you can convert user input into different data types like int, float, and others.

**Converting User Input to Integer (int)**

To convert the user input to an integer, you can use the int() function. This will allow you to perform arithmetic operations or other integer-specific tasks.

**Example:**

age = input("Enter your age: ")

age = int(age) # Convert the input string to an integer

print(f"You are {age} years old.")

**Output (example interaction):**

Enter your age: 25

You are 25 years old.

**Converting User Input to Float (float)**

To convert the user input to a floating-point number, you use the float() function. This is useful when working with decimal numbers.

**Example:**

price = input("Enter the price of the item: ")

price = float(price) # Convert the input string to a float

print(f"The price is ${price:.2f}")

**Output (example interaction):**

Enter the price of the item: 19.99

The price is $19.99

**Converting User Input to Boolean (bool)**

You can also convert user input to a boolean value using bool(). However, you need to manually check if the input represents a True or False value.

**Example (Checking "yes"/"no" as input):**

response = input("Do you want to continue? (yes/no): ")

response = response.lower() == "yes" # Converts to True if "yes", otherwise False

print(f"Continue: {response}")

**Output (example interaction):**

Do you want to continue? (yes/no): yes

Continue: True

If the user enters "yes", it is converted to True; otherwise, it will be False.

**Notes:**

The bool() function itself does not automatically interpret strings like "yes" or "no". You need to manually process the input (as shown above) to map specific inputs to True or False.

**Example with Multiple Data Types:**

You can read multiple types of data in a single program and convert them accordingly.

# Getting user input for name (string), age (int), and weight (float)

name = input("Enter your name: ")

age = int(input("Enter your age: "))

weight = float(input("Enter your weight (in kg): "))

print(f"Name: {name}")

print(f"Age: {age} years")

print(f"Weight: {weight} kg")

**Output (example interaction):**

Enter your name: John

Enter your age: 30

Enter your weight (in kg): 70.5

Name: John

Age: 30 years

Weight: 70.5 kg

**Error Handling for Invalid Input:**

When converting user input to a different data type, it’s a good practice to handle errors that may occur if the user enters invalid data (such as non-numeric input when a number is expected).

**Example with Error Handling:**

try:

age = int(input("Enter your age: ")) # Convert input to an integer

height = float(input("Enter your height (in meters): ")) # Convert to float

print(f"Age: {age}, Height: {height} meters.")

except ValueError:

print("Invalid input! Please enter valid numbers.")

**Output (example interaction when invalid input is entered):**

Enter your age: abc

Invalid input! Please enter valid numbers.

In this case, the program uses try-except to catch ValueError if the user enters something that cannot be converted to int or float.

**Opening and** **Closing Files**

1. **Opening files in different modes ('r', 'w', 'a', 'r+', 'w+').**

**ANS:** In Python, files can be opened in different modes depending on what you want to do with the file (e.g., reading, writing, appending). The built-in open() function is used to open a file, and it takes two arguments:

**File name**: The path to the file you want to open.

**Mode**: The mode in which you want to open the file.

Here are the commonly used modes in Python for opening files:

**1. 'r' – Read Mode**

**Description**: This is the default mode. It opens the file for **reading only**. If the file doesn't exist, it raises a FileNotFoundError.

**Usage**: Used when you want to read the contents of an existing file.

**Example:**

# Open the file in read mode

with open('example.txt', 'r') as file:

content = file.read()

print(content)

**2. 'w' – Write Mode**

**Description**: This mode opens the file for **writing**. If the file already exists, it **overwrites** the existing content. If the file doesn't exist, it creates a new one.

**Usage**: Used when you want to create a new file or overwrite an existing file with new data.

**Example:**

# Open the file in write mode

with open('example.txt', 'w') as file:

file.write("Hello, World!\nThis is a new file.")

**3. 'a' – Append Mode**

**Description**: This mode opens the file for **appending**. It adds new content to the end of the file without modifying the existing content. If the file doesn't exist, it creates a new one.

**Usage**: Used when you want to add new data at the end of an existing file.

**Example:**

# Open the file in append mode

with open('example.txt', 'a') as file:

file.write("\nThis is an appended line.")

**4. 'r+' – Read and Write Mode**

**Description**: This mode opens the file for both **reading and writing**. The file must exist. If the file doesn't exist, it raises a FileNotFoundError. It doesn't truncate the file (does not delete content), but you can modify existing content.

**Usage**: Used when you want to read the file's content and then modify it.

**Example:**

# Open the file in read-write mode

with open('example.txt', 'r+') as file:

content = file.read()

print("Current content:", content)

file.seek(0) # Move the file cursor to the beginning

file.write("Updated content.")

**5. 'w+' – Write and Read Mode**

**Description**: This mode opens the file for **both writing and reading**. If the file already exists, it **overwrites** the content of the file. If the file doesn't exist, it creates a new one.

**Usage**: Used when you want to write new data to the file and also read from it.

**Example:**

# Open the file in write-read mode

with open('example.txt', 'w+') as file:

file.write("Hello, this is a new file!")

file.seek(0) # Move the file cursor to the beginning

content = file.read()

print("Content of the file:", content)

**Mode Summary Table:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Mode** | **Description** | **File Exists** | **File Created** | **File Overwritten** | **Notes** |
| 'r' | Read (default) | Yes | No | No | File must exist. |
| 'w' | Write (creates a new file, overwrites existing) | No | Yes | Yes | File is overwritten if exists. |
| 'a' | Append (adds to the file) | Yes | Yes | No | Data is added to the end. |
| 'r+' | Read and Write (does not truncate) | Yes | No | No | File must exist. |
| 'w+' | Write and Read (overwrites file) | No | Yes | Yes | File is overwritten if exists. |
|  |  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**File Opening Best Practices:**

**Using with open()**: It is recommended to use the with statement when opening files. It automatically closes the file once the block of code is executed, even if an error occurs. This prevents potential file handling issues like leaving files open.

with open('example.txt', 'r') as file:

content = file.read()

print(content)

**Error Handling**: Always check if the file exists before attempting to read or write, especially when using 'r' mode. You can use try-except blocks to handle potential errors.

try:

with open('example.txt', 'r') as file:

content = file.read()

print(content)

except FileNotFoundError:

print("File not found.")

**2) Using the open() function to create and access files.**

**ANS:** In Python, the open() function is used to **create** and **access** files. The function allows you to interact with files for various purposes like reading, writing, and appending. Let's break down how you can use open() to create and access files.

Basic Syntax of open()

open(file, mode)

file: The name (and path) of the file you want to access or create.

mode: A string that defines the access mode. This determines whether you want to read, write, or append to the file. Some common modes include:

'r': Read (file must exist).

'w': Write (creates a new file or overwrites an existing file).

'a': Append (creates a new file or appends to an existing file).

'x': Exclusive creation (fails if the file exists).

Creating a File with open()

You can create a new file by using the 'w', 'x', or 'w+' mode. If the file does not exist, it will be created.

1. Using 'w' Mode

The 'w' mode opens a file for writing. If the file doesn't already exist, it will be created. If the file exists, its contents will be overwritten.

# Create a new file (or overwrite if it already exists)

with open('new\_file.txt', 'w') as file:

file.write("This is a newly created file.\n")

file.write("We are writing to this file using Python.")

Effect: The file new\_file.txt is created if it doesn't exist. If it already exists, its contents will be erased and replaced with the new data.

2. Using 'x' Mode

The 'x' mode opens the file for exclusive creation. If the file already exists, the operation will fail and raise a FileExistsError. This mode is useful when you want to ensure that you don't accidentally overwrite an existing file.

try:

# Create a new file (fails if the file already exists)

with open('new\_file.txt', 'x') as file:

file.write("This is a new file created with 'x' mode.")

except FileExistsError:

print("File already exists, cannot create the file.")

Effect: If new\_file.txt already exists, the program will raise a FileExistsError. If it doesn't exist, the file will be created.

Accessing an Existing File with open()

You can use open() to access an existing file in different modes depending on whether you want to read or modify the file.

1. Using 'r' Mode (Read Mode)

The 'r' mode is used for reading a file. The file must already exist; otherwise, it will raise a FileNotFoundError.

# Open an existing file for reading

with open('existing\_file.txt', 'r') as file:

content = file.read()

print(content)

Effect: Opens existing\_file.txt for reading. If the file doesn't exist, Python will throw an error.

2. Using 'w+' Mode (Write and Read Mode)

The 'w+' mode allows both reading and writing to the file. If the file exists, it will be overwritten. If the file doesn't exist, it will be created.

# Open a file for reading and writing

with open('existing\_file.txt', 'w+') as file:

file.write("This is a new content.\n")

file.seek(0) # Move cursor back to the beginning of the file

content = file.read()

print(content)

Effect: The file is opened for both reading and writing. If the file already exists, it will be overwritten, and if it doesn't exist, it will be created.

3. Using 'a' Mode (Append Mode)

The 'a' mode opens the file for appending. If the file exists, new content is added at the end of the file. If the file doesn't exist, it will be created.

# Open a file for appending

with open('existing\_file.txt', 'a') as file:

file.write("\nThis is an appended line of text.")

Effect: Adds new text to the end of existing\_file.txt. If the file doesn't exist, it will be created.

Full Example:

Here’s a complete example that shows how to create, write, read, and append to a file:

# Create a new file and write some text

with open('example.txt', 'w') as file:

file.write("This is an example file.\n")

file.write("We are demonstrating the open() function.\n")

# Read the contents of the file

with open('example.txt', 'r') as file:

content = file.read()

print("File content after creation:")

print(content)

# Append new content to the file

with open('example.txt', 'a') as file:

file.write("Appending more content to the file.\n")

# Read the updated file content

with open('example.txt', 'r') as file:

updated\_content = file.read()

print("\nFile content after appending:")

print(updated\_content)

Output:

File content after creation:

This is an example file.

We are demonstrating the open() function.

File content after appending:

This is an example file.

We are demonstrating the open() function.

Appending more content to the file.

1. **Closing files using close().**

**ANS:** In Python, when you open a file using the open() function, the file is kept open for reading, writing, or appending. Once you are done working with the file, it’s important to **close** it using the close() method. Closing a file ensures that any changes you’ve made to the file are saved properly, and it releases the system resources that were being used.

**Why Close Files?**

**Save Changes**: When you write to a file, the data might not be saved immediately to the disk. Closing the file ensures that all buffered data is written to the file.

**Resource Management**: Keeping files open consumes system resources. It’s good practice to close files to free those resources.

**Prevent Data Loss**: If a file is not closed properly, you might lose some data or face issues in accessing the file later.

**Basic Syntax for Closing a File**

file.close()

After opening the file using open(), you can call the close() method on the file object to close it.

**Example of Using close()**

Here’s a simple example of opening, writing, and then closing a file:

# Open a file in write mode

file = open('example.txt', 'w')

# Write some text to the file

file.write("This is a simple text file.\n")

file.write("We are demonstrating the use of close().")

# Close the file after writing

file.close()

# Now the file is closed, and you can't write or read from it anymore.

In this example:

The file example.txt is opened in **write mode**.

We write two lines of text to the file.

The close() method is called to close the file after writing.

**Key Points to Remember:**

**Always close files after opening them**: This is good practice to avoid resource leakage and ensure data integrity.

**Once a file is closed, it can no longer be accessed** for reading or writing unless it is opened again.

**Example Showing Potential Problem Without close()**

If you forget to close a file, the data might not be written to the disk properly. For example:

# Open a file without closing it properly

file = open('example.txt', 'w')

file.write("This file may not be saved properly if not closed.")

# If the program ends here without closing the file, the content may not be saved.

In this case, because the file was never explicitly closed, it might not save the content to the disk. Python will automatically close the file when the program finishes, but relying on that is not a good practice.

**Best Practice: Using with open() for Automatic File Closure**

Rather than manually calling file.close(), a better practice is to use the with statement when working with files. This ensures that the file is automatically closed as soon as the block of code is executed, even if an error occurs.

**Using with Statement (Recommended Way)**

# Using 'with' ensures the file is closed automatically

with open('example.txt', 'w') as file:

file.write("This file is automatically closed after the block.")

# No need to call file.close(), it is handled automatically.

In this case, the file is opened for writing, and once the with block is finished, the file is automatically closed. This prevents potential errors and makes the code cleaner and more reliable.

**Reading and Writing Files**

1. **Reading from a file using read(), readline(), readlines().**

**ANS:** In Python, you can read files using the methods read(), readline(), and readlines(). Each of these methods reads a file in a different way. Here's a brief explanation of each method and how they work:

**1. read()**

The read() method reads the entire content of the file as a single string. You can also specify the number of bytes to read by passing an argument.

Example**:**

# Open a file in read mode

with open('example.txt', 'r') as file:

content = file.read() # Reads the entire file

print(content)

Use case: This is useful when you want to read the entire file into memory at once.

2**. readline()**

The readline() method reads a single line from the file. Each time it's called, it will return the next line. When it reaches the end of the file, it will return an empty string.

Example:

# Open a file in read mode

with open('example.txt', 'r') as file:

line = file.readline() # Reads the first line

while line:

print(line, end='') # Prints each line without adding an extra newline

line = file.readline() # Reads the next line

Use case: This is useful when you want to process the file one line at a time (for example, when working with large files that don't fit into memory).

**3. readlines()**

The readlines() method reads the entire file and returns a list of lines. Each line in the file becomes an element in the list.

Example:

# Open a file in read mode

with open('example.txt', 'r') as file:

lines = file.readlines() # Reads all lines into a list

for line in lines:

print(line, end='') # Prints each line

Use case: This is useful when you want to work with all the lines in a file but still have the ability to access them individually as a list.

Summary:

**read()**: Reads the entire file at once.

**readline()**: Reads one line at a time.

**readlines()**: Reads all lines and returns them as a list.

Let me know if you need more details or examples!

**2) Writing to a file using write() and writelines().**

**ANS:** In Python, you can write to a file using the methods write() and writelines(). Each method serves a slightly different purpose when writing data to a file.

1. write()

The write() method writes a string to a file. It does not add a newline character automatically after each write, so if you want to write multiple lines, you need to include newline characters (\n) yourself.

Example:

# Open a file in write mode ('w')

with open('output.txt', 'w') as file:

file.write("Hello, this is the first line.\n")

file.write("This is the second line.\n")

file.write("This is the third line.\n")

**Behavior**:

If the file already exists, it will be **overwritten**.

If the file does not exist, it will be **created**.

**Use case**: This is useful when you need to write specific content to a file, like appending strings or writing a small piece of data.

**2. writelines()**

The writelines() method writes a list of strings to a file. Unlike write(), it doesn't automatically add newline characters between the strings, so you need to ensure that each string in the list has a newline character (\n) at the end if you want each item to be written on a new line.

**Example:**

# Open a file in write mode ('w')

with open('output.txt', 'w') as file:

lines = ["Hello, this is the first line.\n",

"This is the second line.\n",

"This is the third line.\n"]

file.writelines(lines)

**Behavior**:

As with write(), if the file exists, it will be overwritten.

If the file doesn't exist, it will be created.

**Use case**: This is useful when you have a collection of strings (like a list or a tuple) and want to write them all at once.

**Summary:**

**write()**: Writes a single string to a file. You need to manually add newlines (\n) if needed.

**writelines()**: Writes a list of strings to a file. Like write(), it doesn't automatically add newlines between the strings, so each string must have a newline character (\n) if you want to write each string on a new line.

Let me know if you need further clarification or examples!

**Exception Handling**

**1)Introduction to exceptions and how to handle them using try, except, and finally.**

**ANS:** Exceptions are events or errors that disrupt the normal flow of a program. These errors can arise due to various reasons, such as invalid user input, problems with external systems (like databases or files), or division by zero. When an exception occurs, the program stops executing unless the error is handled.

Python provides a way to handle these errors using **exceptions**. The mechanism allows you to "catch" errors and handle them gracefully, without stopping the entire program. You can also define your own exceptions if necessary.

**Basic Structure of Exception Handling in Python**

The primary way to handle exceptions in Python is by using the try, except, and finally blocks.

**1. try Block**

The try block is where you write the code that might raise an exception. If an exception occurs within the try block, Python will stop executing that block and transfer control to the corresponding except block.

**2. except Block**

The except block is used to catch the exception and handle it. You can specify the type of exception you want to catch (e.g., ZeroDivisionError, FileNotFoundError). If no specific exception is mentioned, the except block will catch all exceptions.

**3. finally Block**

The finally block is optional and is used to execute code that should run regardless of whether an exception was raised or not. It's commonly used for cleanup actions, such as closing files or releasing resources.

**Basic Example: Handling Exceptions**

Here is an example that demonstrates how to handle a simple exception:

try:

# Attempt to divide by zero

result = 10 / 0

except ZeroDivisionError:

print("Error: You cannot divide by zero!")

finally:

print("This will always be executed.")

**Explanation:**

The try block contains code that might raise an exception (in this case, dividing by zero).

The except ZeroDivisionError block catches the ZeroDivisionError exception and prints an error message.

The finally block always executes, regardless of whether an exception was raised or not.

**More Detailed Example: Catching Multiple Exceptions**

You can handle different types of exceptions with multiple except blocks:

try:

# Read a number from the user

num = int(input("Enter a number: "))

result = 10 / num

except ValueError:

print("Error: That's not a valid number!")

except ZeroDivisionError:

print("Error: You cannot divide by zero!")

except Exception as e:

print(f"An unexpected error occurred: {e}")

finally:

print("Execution completed.")

**Explanation:**

**ValueError**: Catches cases where the user enters non-numeric input.

**ZeroDivisionError**: Catches the case where the user enters 0, which would result in a division by zero.

**Exception**: This is a generic exception handler that will catch any other unexpected errors not already handled by the previous except blocks.

**Key Points:**

**try**: Code that might raise an exception.

**except**: Catches and handles the exception.

**finally**: Executes code that must always run (such as cleanup code).

You can catch multiple specific exceptions and also use a generic except Exception block for unexpected errors.

**Best Practices for Exception Handling:**

**Be Specific**: Catch specific exceptions whenever possible. This helps in debugging and understanding what went wrong.

**Avoid Bare except**: Avoid using a generic except block without specifying the exception type, as this can hide bugs or issues in your code.

**Log Errors**: Instead of just printing errors, it's a good practice to log them for later review, especially in larger applications.

**Clean-up with finally**: Use the finally block for resource cleanup (e.g., closing files or database connections), ensuring it runs regardless of an error.

Let me know if you'd like further examples or details!

1. **Understanding multiple exceptions and custom exceptions.**

**ANS:** In Python, you can handle multiple exceptions in a single try block by using multiple except blocks. This allows you to handle different types of errors in specific ways. You can also use the else block to run code if no exceptions occur.

**1. Multiple except Blocks**

Each except block can handle a different type of exception. If multiple exceptions could potentially arise from the same try block, you can handle each exception in its respective except block.

**Example:**

try:

x = int(input("Enter a number: "))

y = 10 / x # Potential ZeroDivisionError or ValueError

except ZeroDivisionError:

print("Error: You cannot divide by zero.")

except ValueError:

print("Error: Invalid input. Please enter a valid integer.")

except Exception as e: # Catch any other exceptions

print(f"An unexpected error occurred: {e}")

else:

print(f"Result: {y}")

finally:

print("Execution completed.")

**Explanation:**

**ZeroDivisionError**: Handles the case where the user inputs 0 and the program tries to divide by zero.

**ValueError**: Handles the case where the user inputs a non-integer value that causes the conversion to int() to fail.

**Exception**: Catches any other errors that aren't specifically handled by the previous except blocks.

**else**: Executes if no exceptions occur, printing the result.

**finally**: Runs regardless of whether an exception occurred or not, useful for cleanup.

**2. Handling Multiple Exceptions in a Single except Block**

Instead of having separate except blocks for each exception, you can catch multiple exceptions in a single except block by using a tuple.

**Example:**

try:

x = int(input("Enter a number: "))

y = 10 / x # Potential ZeroDivisionError or ValueError

except (ZeroDivisionError, ValueError) as e:

print(f"Error occurred: {e}")

else:

print(f"Result: {y}")

finally:

print("Execution completed.")

**Explanation:**

The except block catches both ZeroDivisionError and ValueError and prints the error message. The variable e holds the exception instance, so we can display the error message.

**Custom Exceptions in Python**

In Python, you can also define your own custom exceptions. Custom exceptions are useful when you want to handle specific errors that are relevant to your application or domain. Custom exceptions should usually be defined as classes that inherit from Python’s built-in Exception class.

**3. Defining Custom Exceptions**

You can define a custom exception by creating a new class that inherits from the built-in Exception class (or any of its subclasses).

**Example:**

# Define a custom exception

class NegativeValueError(Exception):

"""Exception raised for negative values."""

def \_\_init\_\_(self, value):

self.value = value

self.message = f"Invalid input: {value} is negative."

super().\_\_init\_\_(self.message)

# Function that raises custom exception

def process\_value(value):

if value < 0:

raise NegativeValueError(value)

else:

return f"Processing value: {value}"

# Using the custom exception

try:

result = process\_value(-5)

print(result)

except NegativeValueError as e:

print(e)

**Explanation:**

We define a custom exception NegativeValueError by inheriting from the base Exception class.

The \_\_init\_\_ method takes a value and constructs an error message.

In the process\_value function, we raise the custom exception if the value is negative.

The try-except block handles the custom exception and prints the message.

**4. Using Custom Exceptions with Multiple Exceptions**

You can combine custom exceptions with built-in exceptions to handle specific errors in a more granular way.

**Example:**

class NegativeValueError(Exception):

"""Exception raised for negative values."""

def \_\_init\_\_(self, value):

self.value = value

self.message = f"Negative value error: {value} is invalid."

super().\_\_init\_\_(self.message)

class TooLargeValueError(Exception):

"""Exception raised for values that are too large."""

def \_\_init\_\_(self, value):

self.value = value

self.message = f"Too large value error: {value} exceeds limit."

super().\_\_init\_\_(self.message)

def process\_value(value):

if value < 0:

raise NegativeValueError(value)

elif value > 100:

raise TooLargeValueError(value)

return f"Value {value} is processed successfully."

try:

result = process\_value(200)

print(result)

except NegativeValueError as e:

print(e)

except TooLargeValueError as e:

print(e)

**Explanation:**

We define two custom exceptions: NegativeValueError and TooLargeValueError.

The process\_value function checks if the value is negative or too large and raises the corresponding exception.

The try-except block handles both custom exceptions separately.

**Summary:**

**Multiple Exceptions**: You can handle multiple exceptions using multiple except blocks or by grouping exceptions in a tuple.

**Custom Exceptions**: You can define your own exceptions by creating a class that inherits from Python's built-in Exception class, allowing you to raise and catch domain-specific errors.

**Best Practices**:

Always catch specific exceptions first before catching more general exceptions.

Use custom exceptions when your application needs to handle domain-specific errors in a more controlled manner.

Keep the try block as small as necessary to avoid catching unintended exceptions.

Let me know if you need further clarification or additional examples!

**Class and** **Object (OOP Concepts)**

**1)Understanding the concepts of classes, objects, attributes, and methods in Python.**

**ANS:** Understanding the Concepts of Classes, Objects, Attributes, and Methods in Python

In Python, **object-oriented programming (OOP)** is a paradigm that organizes code into **classes** and **objects**. These concepts help structure code in a way that is easier to manage and understand. Let's break down the key components of OOP: **classes**, **objects**, **attributes**, and **methods**.

**1. Classes in Python**

A **class** is like a blueprint or template for creating objects. It defines the structure and behavior that the objects created from it will have. A class can contain **attributes** (variables) and **methods** (functions).

**Class Definition**: A class is defined using the class keyword in Python.

**Example of Class:**

class Dog:

# Constructor to initialize the object's attributes

def \_\_init\_\_(self, name, breed):

self.name = name # Attribute

self.breed = breed # Attribute

# Method to display dog information

def bark(self):

print(f"{self.name} says woof!")

# Creating an instance (object) of the Dog class

dog1 = Dog("Rex", "Golden Retriever")

# Accessing attributes and calling methods

print(dog1.name) # Output: Rex

dog1.bark() # Output: Rex says woof!

**Explanation:**

**Class Definition**: class Dog: defines the class Dog.

**Constructor**: The \_\_init\_\_ method is a special method called a constructor, which initializes the attributes of an object when it is created. self is a reference to the current instance of the class.

**Attributes**: self.name and self.breed are attributes of the class. These store data about the object.

**Method**: bark() is a method that defines behavior for the class. It is a function that is associated with the object and can be called using dog1.bark().

**2. Objects in Python**

An **object** is an instance of a class. Once a class is defined, you can create many objects (instances) from that class. Each object has its own unique set of attributes.

**Example:**

dog2 = Dog("Bella", "Bulldog")

print(dog2.name) # Output: Bella

dog2.bark() # Output: Bella says woof!

**Explanation:**

dog2 = Dog("Bella", "Bulldog") creates a new object dog2 of the class Dog. It has its own attributes and can call the class's methods independently of other objects.

**3. Attributes in Python**

**Attributes** are variables that hold data or state about an object. In Python, attributes are defined inside a class, usually in the constructor (\_\_init\_\_). Attributes are accessed using dot notation.

**Instance Attributes**: These are attributes that are specific to an object. Each object can have different values for these attributes.

**Class Attributes**: These are attributes that are shared among all instances of a class. They are defined directly within the class.

**Example:**

class Dog:

species = "Canis familiaris" # Class attribute (shared by all instances)

def \_\_init\_\_(self, name, breed):

self.name = name # Instance attribute

self.breed = breed # Instance attribute

# Creating instances of the Dog class

dog1 = Dog("Rex", "Golden Retriever")

dog2 = Dog("Bella", "Bulldog")

# Accessing instance and class attributes

print(dog1.species) # Output: Canis familiaris

print(dog2.species) # Output: Canis familiaris

print(dog1.name) # Output: Rex

print(dog2.name) # Output: Bella

**Explanation:**

**Instance Attribute**: self.name and self.breed are instance attributes. They are unique to each object (e.g., dog1 and dog2).

**Class Attribute**: species is a class attribute. It is shared across all instances of the class and remains the same for every object.

**4. Methods in Python**

**Methods** are functions that are defined inside a class and describe the behavior or actions that an object can perform. Methods typically operate on the attributes of the object (instance methods) and are accessed using the dot notation.

**Types of Methods:**

**Instance Methods**: These are regular methods that operate on the instance (object) attributes.

**Class Methods**: These are methods that operate on class-level attributes and are defined using the @classmethod decorator.

**Static Methods**: These are methods that do not operate on instance or class attributes and are defined using the @staticmethod decorator.

**Example of Methods:**

class Dog:

species = "Canis familiaris"

def \_\_init\_\_(self, name, breed):

self.name = name

self.breed = breed

def bark(self): # Instance method

print(f"{self.name} says woof!")

@classmethod

def species\_info(cls): # Class method

print(f"The species is {cls.species}")

@staticmethod

def common\_fact(): # Static method

print("Dogs are loyal animals.")

# Creating an instance of Dog

dog1 = Dog("Rex", "Golden Retriever")

# Calling instance method

dog1.bark() # Output: Rex says woof!

# Calling class method

dog1.species\_info() # Output: The species is Canis familiaris

# Calling static method

dog1.common\_fact() # Output: Dogs are loyal animals.

**Explanation:**

**Instance Method (bark)**: This method uses self to access instance attributes and performs an action (barking).

**Class Method (species\_info)**: This method uses cls to access class-level attributes and can be called on both the class and instances.

**Static Method (common\_fact)**: This method does not require access to either instance or class attributes, so it doesn't take self or cls as arguments. It can be called directly from the class or the object.

**2)Difference between local and global variables.**

**ANS:** In Python, **local** and **global variables** are two types of variables that differ based on their **scope** and **lifetime**.

1. Local Variables

A local variable is a variable that is defined inside a function or block and can only be accessed within that function or block. The scope of a local variable is limited to the function or block in which it is defined. Once the function call completes, the local variable is destroyed.

Scope: The region of the code where the variable is accessible (inside the function or block).

Lifetime: The time period during which the variable exists (only while the function is executing).

Example of Local Variable:

def greet():

name = "Alice" # Local variable

print("Hello,", name)

greet() # Output: Hello, Alice

# print(name) # This will raise an error because `name` is local to `greet()`

Explanation:

The variable name is defined inside the function greet(), so it is a local variable.

You can access name only inside the greet() function.

Trying to access name outside the function will result in an error because it's out of scope.

2. Global Variables

A global variable is a variable that is defined outside any function or class and can be accessed anywhere in the code, including inside functions. Global variables are available throughout the entire script or module.

Scope: The region of the code where the variable is accessible (anywhere in the program after its definition).

Lifetime: The variable exists as long as the program runs, i.e., it is available from the point of definition until the program terminates.

Example of Global Variable:

name = "Bob" # Global variable

def greet():

print("Hello,", name) # Accessing the global variable

greet() # Output: Hello, Bob

print(name) # Output: Bob

Explanation:

The variable name is defined outside any function, making it a global variable.

It can be accessed both inside the greet() function and outside the function.

3. Key Differences Between Local and Global Variables

|  |  |  |
| --- | --- | --- |
| Feature | Local Variable | Global Variable |
| Definition | Defined inside a function or block. | Defined outside any function or class. |
| Scope | Limited to the function or block where defined. | Accessible throughout the entire program (after definition). |
| Lifetime | Exists only during the execution of the function or block. | Exists throughout the lifetime of the program. |
| Access | Can only be accessed within the function or block. | Can be accessed from anywhere in the program. |
| Modification | Can be modified within the function (if not declared global). | Can be modified from anywhere, unless restricted. |
| Memory | Memory is allocated when the function is called and released when it ends. | Memory is allocated when the program starts and released when the program ends. |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

4. Modifying Global Variables Inside a Function

By default, a function treats any variable assigned to inside it as a local variable. If you want to modify a global variable inside a function, you need to explicitly declare it as global.

Example: Modifying Global Variable Inside a Function

counter = 0 # Global variable

def increment():

global counter # Declare `counter` as a global variable to modify it

counter += 1

increment()

increment()

print(counter) # Output: 2

Explanation:

Without the global keyword, the counter inside the function would be treated as a local variable.

The global keyword tells Python to use the counter variable defined outside the function, and we modify its value.

5. Global Variables and Best Practices

While global variables are useful, excessive use of them is generally discouraged in large programs for the following reasons:

Difficulty in debugging: It can be hard to track the changes to global variables, especially in large programs.

Unintended side effects: Since global variables can be accessed and modified anywhere in the program, they can lead to unintended behavior if not managed carefully.

Example of Both Local and Global Variables:

counter = 10 # Global variable

def increment():

counter = 5 # Local variable (this does NOT modify the global variable)

counter += 1

print("Inside function:", counter)

increment() # Output: Inside function: 6

print("Outside function:", counter) # Output: Outside function: 10

Explanation:

Inside the increment() function, counter is a local variable that shadows the global counter.

The global counter remains unchanged because the function only modifies the local counter.

**Inheritance**

**1)Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python.**

**ANS:** In Python, inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called **derived classes**, and the existing classes are **base classes**. Python supports different types of inheritance:

**1. Single Inheritance**

A child class inherits from one parent class.

class Parent:

def greet(self):

print("Hello from Parent")

class Child(Parent):

def welcome(self):

print("Welcome from Child")

obj = Child()

obj.greet()

obj.welcome()

**Output:**

Hello from Parent

Welcome from Child

**2. Multilevel Inheritance**

A child class inherits from a class which itself inherits from another class.

class Grandparent:

def greet(self):

print("Hello from Grandparent")

class Parent(Grandparent):

def hello(self):

print("Hello from Parent")

class Child(Parent):

def welcome(self):

print("Welcome from Child")

obj = Child()

obj.greet()

obj.hello()

obj.welcome()

**Output:**

Hello from Grandparent

Hello from Parent

Welcome from Child

**3. Multiple Inheritance**

A class inherits from more than one parent class.

class Father:

def skills(self):

print("Father: Cooking")

class Mother:

def hobbies(self):

print("Mother: Dancing")

class Child(Father, Mother):

def show(self):

print("Child: Learning")

obj = Child()

obj.skills()

obj.hobbies()

obj.show()

**Output:**

Father: Cooking

Mother: Dancing

Child: Learning

**4. Hierarchical Inheritance**

Multiple child classes inherit from the same parent class.

class Parent:

def greet(self):

print("Hello from Parent")

class Child1(Parent):

def child1\_func(self):

print("Child1 function")

class Child2(Parent):

def child2\_func(self):

print("Child2 function")

obj1 = Child1()

obj2 = Child2()

obj1.greet()

obj2.greet()

**Output:**

Hello from Parent

Hello from Parent

**5. Hybrid Inheritance**

A combination of two or more types of inheritance. Python uses the **Method Resolution Order (MRO)** to manage this.

class A:

def method(self):

print("Class A")

class B(A):

def method\_b(self):

print("Class B")

class C(A):

def method\_c(self):

print("Class C")

class D(B, C):

def method\_d(self):

print("Class D")

obj = D()

obj.method() # Will use MRO to find which 'method' to call

**Output (based on MRO):**

Class A

Want me to show diagrams or visuals for these? They can make it even clearer.

**2)Using the super() function to access properties of the parent class.**

**ANS:** Great! Let's dive into the super() function in Python.

What is super() in Python?

The super() function is used to call methods or access properties from a parent (superclass) in a child (subclass), especially useful in inheritance. It helps avoid explicitly naming the parent class and supports maintainability and multiple inheritance better.

Basic Syntax:

super().method\_name()

Example 1: Using super() to Call Parent Method

class Parent:

def greet(self):

print("Hello from Parent")

class Child(Parent):

def greet(self):

super().greet() # Call the parent class method

print("Hello from Child")

obj = Child()

obj.greet()

Output:

Hello from Parent

Hello from Child

Example 2: Using super() in \_\_init\_\_() Constructor

class Person:

def \_\_init\_\_(self, name):

self.name = name

print(f"Person: {self.name}")

class Student(Person):

def \_\_init\_\_(self, name, grade):

super().\_\_init\_\_(name) # Call Person's \_\_init\_\_

self.grade = grade

print(f"Student Grade: {self.grade}")

s = Student("Alice", "A")

Output:

Person: Alice

Student Grade: A

Why Use super() Instead of Parent Class Name?

More maintainable when class names change.

Plays well with multiple inheritance (respects MRO).

Cleaner and avoids redundancy.

Example 3: super() with Multiple Inheritance

class A:

def \_\_init\_\_(self):

print("A init")

class B(A):

def \_\_init\_\_(self):

super().\_\_init\_\_()

print("B init")

class C(A):

def \_\_init\_\_(self):

super().\_\_init\_\_()

print("C init")

class D(B, C):

def \_\_init\_\_(self):

super().\_\_init\_\_()

print("D init")

d = D()

Output (based on MRO):

A init

C init

B init

D init

You can check the MRO with:

print(D.\_\_mro\_\_)

Let me know if you want me to explain MRO (Method Resolution Order) or show how super() works under the hood.

**Method Overloading and Overriding**

**1)Method overloading: defining multiple methods with the same name but different parameters.**

ANS: Great! Let’s break down **Method Overloading** in Python. 🧠👇

1) What is Method Overloading?

Method Overloading means having multiple methods with the same name but different parameters in the same class.

In many languages like Java or C++, this is done using different parameter counts/types.

BUT in Python...

Python does NOT support method overloading in the traditional sense because:

Python allows only the last defined method with a given name to exist.

Earlier definitions get overwritten.

Example (NOT true overloading):

class Demo:

def show(self):

print("No arguments")

def show(self, a):

print("One argument:", a)

obj = Demo()

obj.show(10) # Works

# obj.show() # ❌ Error: missing 1 required positional argument

Output:

One argument: 10

Only the second show() method exists — the first one is overwritten.

You can simulate method overloading using:

Default arguments

Variable arguments (\*args, \*\*kwargs)

Type checking (optional)

Example 1: Using Default Arguments

class Demo:

def show(self, a=None):

if a is not None:

print("Argument:", a)

else:

print("No arguments")

obj = Demo()

obj.show() # No arguments

obj.show(100) # Argument: 100

📘 Example 2: Using \*args

class Math:

def add(self, \*args):

return sum(args)

obj = Math()

print(obj.add(1, 2)) # 3

print(obj.add(1, 2, 3, 4)) # 10

Example 3: Type Checking (optional & not common)

class Test:

def greet(self, name=None):

if isinstance(name, str):

print(f"Hello, {name}")

else:

print("Hello!")

obj = Test()

obj.greet() # Hello!

obj.greet("Charlie") # Hello, Charlie

Summary:

| Language | Supports True Overloading |
| --- | --- |
| Java, C++ | ✅ Yes |
| Python | ❌ Not directly |
| Workaround in Python | ✅ With default/variable arguments |

Want to see how method overriding (same method in child class) works next? It’s often confused with overloading.

**2)Method overriding: redefining a parent class method in the child class.**

**ANS**: Absolutely! Let’s look at **Method Overriding** — one of the core concepts of **object-oriented programming** in Python.

2) Method Overriding in Python

Method overriding happens when a child class defines a method with the same name and parameters as a method in its parent class.

🧠 Purpose: To change or extend the behavior of the parent class method.

✅ Basic Example:

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def speak(self):

print("Dog barks")

obj = Dog()

obj.speak()

🟢 Output:

Dog barks

⚠️ The Dog class overrides the speak() method from the Animal class.

🧠 How to Access the Parent Method (if needed)?

Use the super() function:

class Dog(Animal):

def speak(self):

super().speak() # Call parent method

print("Dog barks") # Extend functionality

🟢 Output:

Animal speaks

Dog barks

📘 Real-World Example: Bank Account

class Account:

def interest\_rate(self):

return "Base rate: 5%"

class SavingsAccount(Account):

def interest\_rate(self):

return "Savings rate: 7%"

acc = SavingsAccount()

print(acc.interest\_rate())

🟢 Output:

Savings rate: 7%

✅ Summary of Overriding:

|  |  |
| --- | --- |
| Feature | Description |
| Method Name | Same as parent |
| Parameters | Same as parent |
| Class | Defined in child class |
| Purpose | Customize or extend base class behavior |
| Use of super() | Optional, to call parent version |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Let me know if you'd like to see overriding with constructors (\_\_init\_\_) next, or dive into polymorphism, which is often tied to overriding!

**SQLite3 and PyMySQL (Database Connectors)**

**1)Introduction to SQLite3 and PyMySQL for database connectivity.**

**ANS**: Sure! Here's a quick and clear introduction to **SQLite3** and **PyMySQL**—two popular ways to connect Python to databases. Let’s break it down 👇

**1) SQLite3 in Python**

What is SQLite?

A **lightweight**, **serverless**, and **self-contained** SQL database engine.

Stores the entire database in a **single file**.

Great for **small to medium applications**, local data storage, or testing.

✅ **Built-in in Python** – no extra installation needed!

Example: Connecting with SQLite3

import sqlite3

# Connect to (or create) a database file

conn = sqlite3.connect('mydata.db')

# Create a cursor object

cursor = conn.cursor()

# Create a table

cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY,

name TEXT,

age INTEGER

)

''')

# Insert data

cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ("Alice", 25))

# Commit changes

conn.commit()

# Query data

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

print(cursor.fetchall())

# Close connection

conn.close()

2) **PyMySQL** in Python

What is PyMySQL?

A **pure Python** MySQL client library.

Lets Python connect to a **MySQL** or **MariaDB** database.

You need to have a **MySQL server** running.

Installation:

pip install pymysql

Example: Connecting with PyMySQL

import pymysql

# Connect to MySQL database

conn = pymysql.connect(

host='localhost',

user='your\_username',

password='your\_password',

database='your\_database'

)

# Create a cursor object

cursor = conn.cursor()

# Create table

cursor.execute('''

CREATE TABLE IF NOT EXISTS employees (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

salary FLOAT

)

''')

# Insert data

cursor.execute("INSERT INTO employees (name, salary) VALUES (%s, %s)", ("John", 50000))

# Commit changes

conn.commit()

# Fetch data

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

print(cursor.fetchall())

# Close connection

conn.close()

✅ Key Differences

| **Feature** | **SQLite3** | **PyMySQL** |
| --- | --- | --- |
| Storage | File-based (local) | Server-based (remote/local) |
| Setup | No setup needed | Requires MySQL server |
| Use case | Lightweight apps, prototyping | Web apps, multi-user systems |
| Speed | Faster for small apps | Scales better for large apps |

**2)Creating and executing SQL queries from Python using these connectors.**

**ANS**: Absolutely! Let's walk through how to **create and execute SQL queries** in Python using both **SQLite3** and **PyMySQL** connectors.

1) **Using SQLite3**

SQLite is super convenient for small apps, and Python has built-in support for it via the sqlite3 module.

Step-by-Step: Create & Execute SQL Queries (SQLite3)

import sqlite3

# 1. Connect to the database (creates file if not exists)

conn = sqlite3.connect('example.db')

# 2. Create a cursor object

cursor = conn.cursor()

# 3. Create a table

cursor.execute('''

CREATE TABLE IF NOT EXISTS products (

id INTEGER PRIMARY KEY,

name TEXT NOT NULL,

price REAL

)

''')

# 4. Insert records

cursor.execute("INSERT INTO products (name, price) VALUES (?, ?)", ("Laptop", 1200.50))

cursor.execute("INSERT INTO products (name, price) VALUES (?, ?)", ("Mouse", 25.00))

# 5. Fetch all records

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

rows = cursor.fetchall()

for row in rows:

print(row)

# 6. Update a record

cursor.execute("UPDATE products SET price = ? WHERE name = ?", (1300.00, "Laptop"))

# 7. Delete a record

cursor.execute("DELETE FROM products WHERE name = ?", ("Mouse",))

# 8. Commit and close

conn.commit()

conn.close()

2) **Using PyMySQL**

This lets you connect to a **MySQL** or **MariaDB** server. It’s excellent for web apps and larger-scale systems.

Step-by-Step: Create & Execute SQL Queries (PyMySQL)

import pymysql

# 1. Connect to the MySQL database

conn = pymysql.connect(

host='localhost',

user='your\_username',

password='your\_password',

database='your\_database'

)

# 2. Create a cursor

cursor = conn.cursor()

# 3. Create a table

cursor.execute('''

CREATE TABLE IF NOT EXISTS students (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

grade FLOAT

)

''')

# 4. Insert data

cursor.execute("INSERT INTO students (name, grade) VALUES (%s, %s)", ("Alice", 89.5))

cursor.execute("INSERT INTO students (name, grade) VALUES (%s, %s)", ("Bob", 92.0))

# 5. Fetch all data

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

for row in cursor.fetchall():

print(row)

# 6. Update a record

cursor.execute("UPDATE students SET grade = %s WHERE name = %s", (95.0, "Bob"))

# 7. Delete a record

cursor.execute("DELETE FROM students WHERE name = %s", ("Alice",))

# 8. Commit and close

conn.commit()

conn.close()

✅ Tips:

|  |  |  |
| --- | --- | --- |
| **Task** | **SQLite3** | **PyMySQL** |
| Placeholders | ? | %s |
| Cursor | cursor = conn.cursor() | Same |
| Commit | conn.commit() | Same |
| Fetch data | cursor.fetchall() | Same |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Search and Match Functions**

1. **Using re.search() and re.match() functions in Python’s re module for pattern matching.**

**ANS:** Sure! Let's go over how re.search() and re.match() work in Python’s re (regular expression) module, with examples.

**re.search()**

* **Purpose:** Looks for a match **anywhere** in the string.
* **Returns:** A match object if found, otherwise None.

**Example:**

import re

text = "Hello, my name is ChatGPT."

match = re.search(r"name", text)

if match:

print("Found:", match.group())

else:

print("Not found.")

**Output:**

Found: name

Here, re.search() found "name" in the middle of the string.

**re.match()**

* **Purpose:** Checks for a match **only at the beginning** of the string.
* **Returns:** A match object if the beginning matches, otherwise None.

**Example:**

import re

text = "Hello, my name is ChatGPT."

match = re.match(r"Hello", text)

if match:

print("Found:", match.group())

else:

print("Not found.")

**Output:**

Found: Hello

But if you try this:

match = re.match(r"name", text)

You’ll get:

Not found.

Because "name" is **not** at the start of the string.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

2) **Difference between search and match.**

**ANS:** Sure! Here's a clear breakdown of the **difference between re.search() and re.match()** in Python:

**🔍 re.search() vs 🧭 re.match()**

|  |  |  |
| --- | --- | --- |
| **Feature** | **re.match()** | **re.search()** |
| Where it looks | Only at the **beginning** of the string | **Anywhere** in the string |
| Use case | When you want to check if a string **starts** with a pattern | When you want to find a pattern **anywhere** |
| Returns | Match object if pattern is at the start, else None | Match object if pattern is found, else None |
| Common usage | Input validation (e.g., strings starting with a keyword) | Searching for a keyword or pattern inside a string |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Example Comparison:**

import re

text = "Python is fun"

# re.match()

match\_result = re.match(r"Python", text)

# re.search()

search\_result = re.search(r"fun", text)

print("Match:", match\_result.group() if match\_result else "No match")

print("Search:", search\_result.group() if search\_result else "No search result")

**Output:**

Match: Python

Search: fun

Now try:

re.match(r"fun", text)

**Result:** No match — because fun is **not** at the start of the string.

**In short:**

re.match() → Checks the **start only**.

re.search() → Scans the **entire string**.