PYTHON PROGRAMMING

(SKILL ENHANCEMENT COURSE)

UNTI-I: History of Python Programming Language, Thrust Areas of Python, Installing Anaconda Python Distribution, Installing and Using Jupyter Notebook. Parts of Python Programming Language: Identifiers, Keywords, Statements and Expressions, Variables, Operators, Precedence and Associativity, Data Types, Indentation, Comments, Reading Input, Print Output, Type Conversions, the type () Function and Is Operator, Dynamic and Strongly Typed Language. Control Flow Statements: if statement, if-else statement, if...elif…else, Nested if statement, while Loop, for Loop, continue and break Statements, Catching Exceptions Using try and except Statement.

Answers:-

History of Python Programming Language:

Python was created by Guido van Rossum in the Netherlands in 1990 and was named after the popular British comedy troupe Monty Python's Flying Circus. van Rossum developed Python as a hobby, and Python has become a popular programming language widely used in industry and academia due to its simple, concise, and intuitive syntax and extensive library.

Python was developed by Guido van Rossum in the late eighties at the "National Research Institute for Mathematics and Computer Science" at Netherlands.

Standard C Python interpreter is managed by "Python Software Foundation".

Python born, name picked – Dec 1989

First public release (USENET) – Feb 1991

Python.org website – 1996 or 1997

2.0 released – 2000

Python Software Foundation – 2001

The Python is implemented in other programming libraries as follow.

- JPython for Java

- Iron for C#
- Stackless Python for C
- PyPy for Python itself JIT Compilation.
- RPython for Ruby Programming

Python Editions:

1. Python 1.0 (January 1994)
2. Python 2.0 (October 2000)
3. Python 3.0 (December 2008)
4. Python 3.11.4 (6th June 2023)
5. Python 3.12.5 (7th August 2024)

Standard libraries are written in python itself.

Original motivation for creating python was the perceived need for a higher level language in Amoeba (Operating System) Project. He was used too long C code for System utilities.

What is Python Programming?

Ans:

"Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured, object-oriented and functional programming"

"Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- Web Development (server-side),
- Software Development,
- Mathematics,
- System Scripting.

"

Python is **a high-level, general-purpose, interpreted object-oriented programming language**. **used to build websites and software, automate** tasks, and analyze data, dynamic typed, Strong Typed, Indentation, server-client programming.

Key Features of Python Programming

Here are a few features of Python that make it a popular programming language in today's time.

1. Portable Language

It is a cross-platform language. It can run on Linux, macOS, and Windows. For example, you can run the Python code for Windows in Linux or macOS, too.

2. Standard Library

It offers various modules like operators, mathematical functions, libraries such as NumPy, Pandas, Tensorflow, etc., and packages paving the way for the developers to save time to avoid re-writing the codes from scratch. To provide more functionality and packages, they also provide Python Package Index.

3. High-Level Language

It is a high-level, general-purpose programming language. Unlike machine language like C, C++, It is a human-readable language. In other words, even a layman can understand the programs.

4. Easy to learn and use

It is easy to understand and easy to code, and anyone can learn Python within a few days. For example, a simple Python program to add two numbers is as follows:

```
a = 8
b = 9
print(a+b)
```

We have completed this program within three lines. Whereas in Java, C++ and C, it takes more lines. That is why Python is known as an easy and precise language.

5. Dynamic Language

Declaring the type of a variable is not needed. For example, let us declare an integer number 7 for a variable a. Rather than declare it as:

int a = 7 (this is necessary for statically-typed language like C)
We declare it as
a = 7

But, the programmers have to be careful regarding runtime errors.

6. Extensible Language

Code can be used to compile in C or C++ language so that it can be utilized for our Python code. This is achieved because it converts the program to byte code.

7. Interpreted Language

Line-by-line execution of source code, converted into byte code; thus, compiling the code is not necessary, making it easy to debug if required.

8. Object-Oriented Programming Language

It supports procedural, functional, structural, and also object-oriented language such as abstraction, encapsulation, inheritance, and polymorphism which is considered important by the Python coder.

9. Free and Open-source libraries

It is a free open-source platform that can be downloaded easily from their Official Website.

10. Support is provided for Graphical User Interface

Lots of Graphical User Interface frameworks are available in Python, thus helping the software user, and it sticks to platform-specific technologies. It can be used in software

development, web development, etc

Applications of Python:

Thrust Areas of Python:

Python is a highly versatile and powerful programming language, excelling in a wide range of applications due to its simplicity, flexibility, and strong community support.

1. Web Development

- **Why**: Python's clean and readable syntax, combined with powerful frameworks, makes it ideal for building web applications quickly and efficiently.
- **Tools**: Django, Flask, Pyramid.
- **Applications**: E-commerce sites, content management systems (CMS), RESTful APIs, and social media platforms.
- **Key Features**: Rapid development, security, scalability, and simplicity in handling HTTP requests and responses.
- **Frameworks**: Django for full-stack development, Flask for lightweight applications, Pyramid for flexibility.

2. Data Science and Analytics

- **Why**: Python’s libraries provide comprehensive tools for data manipulation, analysis, and visualization, making it the go-to language for data scientists.
- **Tools**: NumPy, pandas, Matplotlib, seaborn.
- **Applications**: Data analysis, predictive modeling, statistical analysis, and business intelligence.
- **Key Features**: Easy data manipulation, integration with machine learning tools, and extensive support for scientific computing.
- **Frameworks**: Jupyter Notebook for interactive analysis, pandas for data manipulation, Matplotlib for visualization.

3. Machine Learning and Artificial Intelligence

- **Why**: Python’s simplicity and extensive library support make it the language of choice for AI and ML development.
- **Tools**: TensorFlow, Keras, PyTorch, Scikit-learn.
- **Applications**: Image recognition, natural language processing (NLP), recommendation systems, and autonomous systems.
- **Key Features**: Flexibility, support for deep learning, and seamless integration with data processing tools.
- **Frameworks**: TensorFlow and PyTorch for deep learning, Scikit-learn for traditional ML algorithms.

4. Automation and Scripting

- **Why**: Python’s readability and extensive standard library make it perfect for writing scripts to automate repetitive tasks.
- **Tools**: `os`, `sys`, `subprocess`.
- **Applications**: System administration, file manipulation, task automation, and software testing.

- **Key Features**: High-level programming interface, ease of integration with system commands, and rapid development.
- **Frameworks**: Ansible for infrastructure automation, Fabric for SSH task execution.

5. Scientific Computing

- **Why**: Python offers powerful tools for performing complex mathematical computations and simulations.
- **Tools**: SciPy, SymPy.
- **Applications**: Computational physics, chemistry simulations, engineering analysis, and numerical problem-solving.
- **Key Features**: Advanced mathematical functions, symbolic computation, and support for large-scale computations.
- **Frameworks**: SciPy for numerical analysis, SymPy for symbolic mathematics.

6. DevOps and Infrastructure Management

- **Why**: Python's capabilities in automation and scripting are critical in DevOps for managing and deploying infrastructure.
- **Tools**: Ansible, Terraform, SaltStack.
- **Applications**: Continuous integration/continuous deployment (CI/CD), infrastructure as code (IaC), and cloud automation.
- **Key Features**: Automation of repetitive tasks, infrastructure management, and seamless integration with cloud platforms.
- **Frameworks**: Ansible for configuration management, Terraform for cloud infrastructure provisioning.

7. Game Development

- **Why**: Python is ideal for prototyping games and developing simple 2D games due to its easy-to-learn syntax and powerful libraries.
- **Tools**: Pygame, Panda3D.
- **Applications**: 2D games, educational games, and rapid game prototyping.
- **Key Features**: Simple syntax, support for multimedia applications, and strong community support.
- **Frameworks**: Pygame for 2D games, Panda3D for 3D game development.

8. Networking and Cybersecurity

- **Why**: Python’s flexibility and rich set of libraries make it invaluable for network programming and cybersecurity.
- **Tools**: Scapy, Paramiko.
- **Applications**: Network automation, penetration testing, and ethical hacking.
- **Key Features**: Strong support for network protocols, automation of network tasks, and extensive libraries for security operations.
- **Frameworks**: Scapy for network packet manipulation, Paramiko for SSH automation.

9. Education and Academia

- **Why**: Python's readability and simplicity make it an excellent language for teaching programming and computer science concepts.
- **Tools**: IDLE, Jupyter Notebook.
- **Applications**: Programming education, academic research, and educational tools.
- **Key Features**: Easy-to-understand syntax, interactive learning environments, and broad adoption in educational institutions.
- **Frameworks**: Jupyter Notebook for interactive teaching, Tkinter for GUI-based educational tools.

10. Desktop Applications

- **Why**: Python can be used to develop cross-platform desktop applications with graphical user interfaces.
- **Tools**: Tkinter, PyQt, Kivy.
- **Applications**: Simple desktop tools, cross-platform applications, and prototyping.
- **Key Features**: Cross-platform compatibility, rapid development, and rich GUI libraries.
- **Frameworks**: Tkinter for basic GUIs, PyQt for complex interfaces, Kivy for touch-based applications.

11. Internet of Things (IoT)

- **Why**: Python's simplicity and hardware integration capabilities make it ideal for IoT development.
- **Tools**: MicroPython, CircuitPython.
- **Applications**: Embedded systems, home automation, sensor data processing, and robotics.
- **Key Features**: Lightweight and efficient for microcontrollers, strong support for hardware interfaces, and rapid prototyping.
- **Frameworks**: MicroPython for embedded systems, CircuitPython for sensor data management.

12. Financial Technology (FinTech)

- **Why**: Python's powerful data processing and financial libraries make it a natural fit for the FinTech industry.
- **Tools**: QuantLib, Pandas.
- **Applications**: Algorithmic trading, financial modeling, risk management, and payment systems.
- **Key Features**: High-level financial computations, integration with financial APIs, and robust data analysis tools.
- **Frameworks**: QuantLib for financial modeling, Pandas for data analysis.

13. Cloud Computing

- **Why**: Python is deeply integrated with cloud platforms, making it a key language for cloud-native application development.
- **Tools**: Boto3, Google Cloud SDK, Azure SDK.
- **Applications**: Cloud automation, serverless architectures, and cloud-native app development.
- **Key Features**: Seamless integration with cloud services, automation of cloud tasks, and support for serverless computing.
- **Frameworks**: Boto3 for AWS automation, Azure SDK for Microsoft Azure integration.

14. Blockchain and Cryptocurrencies

- **Why**: Python's clarity and strong security features make it suitable for developing blockchain applications and cryptocurrency systems.
- **Tools**: PyCryptodome, Web3.py.
- **Applications**: Smart contracts, decentralized applications (DApps), and cryptocurrency platforms.
- **Key Features**: Strong cryptography libraries, ease of developing blockchain algorithms, and robust security.
- **Frameworks**: Web3.py for Ethereum-based DApps, PyCryptodome for cryptographic functions.

15. Robotics

- **Why**: Python's ability to handle complex algorithms and real-time data processing makes it a key language in robotics.
- **Tools**: Robot Operating System (ROS), PyRobot.
- **Applications**: Robot control systems, sensor integration, and simulation.
- **Key Features**: Real-time data processing, strong support for hardware integration, and easy prototyping.
- **Frameworks**: ROS for robot programming, PyRobot for simplified robotics development.

16. Media and Entertainment

- **Why**: Python's versatility in handling multimedia data makes it a valuable tool in the media and entertainment industry.
- **Tools**: Blender, OpenCV.
- **Applications**: Video editing, animation, special effects, and media automation.
- **Key Features**: Integration with media processing libraries, automation of media workflows, and cross-platform compatibility.
- **Frameworks**: Blender for 3D animation and video editing, OpenCV for image and video processing.

17. Bioinformatics

- **Why**: Python's powerful libraries and tools make it a preferred language for bioinformatics and computational biology.
- **Tools**: Biopython, PyMOL.
- **Applications**: Genomic data analysis, protein structure prediction, and drug discovery.
- **Key Features**: Extensive support for biological data processing, integration with scientific tools, and strong visualization capabilities.
- **Frameworks**: Biopython for biological data analysis, PyMOL for molecular visualization.

Installing Python on Windows/Mac/Linux:

To installing Python on different operating systems:

Windows

1. **Download Python**:
   - Go to the official Python website.
   - Download the latest Python installer for Windows.

2. **Run the Installer**:
   - Double-click the downloaded installer.
   - Check the box that says "Add Python to PATH."
   - Click "Install Now" to start the installation process.

3. **Verify Installation**:
   - Open Command Prompt.
   - Type `python --version` and press Enter. You should see the Python version installed.

macOS

1. **Download Python**:
   - Visit the official Python website.
   - Download the latest macOS installer package.

2. **Run the Installer**:
   - Open the downloaded `.pkg` file.
   - Follow the prompts to install Python.

3. **Verify Installation**:
   - Open Terminal.
   - Type `python3 --version` and press Enter. You should see the Python version installed.

Linux

1. **Using Package Manager**:
   - Most Linux distributions come with Python pre-installed. You can check by running `python3 --version` in the terminal.
   - To install or update Python, use your distribution's package manager. For example:
     - **Debian/Ubuntu**: `sudo apt-get update && sudo apt-get install python3`
     - **Fedora**: `sudo dnf install python3`
     - **Arch**: `sudo pacman -S python`

2. **Verify Installation**:
   - Open Terminal.
   - Type `python3 --version` and press Enter. You should see the Python version installed.

Additional Steps

- **PIP (Python Package Installer)**: Typically, the Python installer includes `pip` (Python's package installer). You can verify `pip` by typing `pip --version` or `pip3 --version` in the terminal or command prompt.
- **Virtual Environments**: To manage dependencies for different projects, consider using virtual environments. You can create a virtual environment with `python -m venv myenv` and activate it with `source myenv/bin/activate` (Unix) or `myenv\Scripts\activate` (Windows).

Following these steps should help you successfully install Python on your system.

Installing Anaconda Python Distribution:

Anaconda is a popular open-source distribution of Python and R, primarily used for scientific computing, data science, machine learning, and large-scale data processing. It simplifies package management and deployment, making it an excellent choice for beginners and professionals alike. Here's how to install Anaconda on your system.

Step 1: Download Anaconda

1. **Go to the Official Anaconda Website:**
   - Visit Anaconda's official download page to find the latest version of Anaconda.

2. **Choose Your Operating System:**

- Select the appropriate installer for your operating system (Windows, macOS, or Linux).

3. **Select Python Version:**
   - Choose the Python version you want to install (typically Python 3.x is recommended).

4. **Download the Installer:**
   - Click the download button to get the installer file.

Step 2: Install Anaconda

For Windows:

1. **Run the Installer:**
   - Locate the downloaded `.exe` file and double-click it to start the installation process.

2. **Follow the Setup Instructions:**
   - Choose "Install for me only" or "All users," depending on your preference.
   - Select the installation path (the default path is usually fine).
   - Check the option to "Add Anaconda to my PATH environment variable" (optional but recommended).
   - Check the option to "Register Anaconda as my default Python 3.x" (recommended).

3. **Finish Installation:**
   - Click "Install" and wait for the installation process to complete.

For macOS:

1. **Run the Installer:**
   - Locate the downloaded `.pkg` file and double-click it to start the installation.

2. **Follow the Setup Instructions:**
   - Click "Continue" through the setup process, selecting the default options unless you need specific configurations.

3. **Finish Installation:**
   - After installation, open the Terminal and type `conda` to verify the installation.

For Linux:

1. **Open Terminal:**
   - Navigate to the directory where the installer script is located.

2. **Run the Installer Script:**
    - Use the command:

      ```
      bash Anaconda3-<version>-Linux-x86_64.sh
      ```

    - Replace `<version>` with the actual version number of the downloaded installer.

3. **Follow the Setup Instructions:**
    - Press `Enter` to review the license agreement, then type `yes` to accept it.
    - Choose the installation location (default is `~/anaconda3`).

4. **Initialize Anaconda:**
    - After installation, run the following command to initialize Anaconda:

      ```
      source ~/.bashrc
      ```

Step 3: Verify the Installation

- Open your terminal (or Anaconda Prompt on Windows).
- Type the following command to check the installation:

  ```
  conda --version
  ```

- You should see the version of Conda installed, confirming that Anaconda is set up.

Step 4: Launch Anaconda Navigator (Optional)

Anaconda Navigator is a graphical user interface (GUI) that allows you to manage packages, environments, and access different tools.

- **Windows/Mac:** You can launch Anaconda Navigator from the start menu or applications folder.
- **Linux:** Run the following command in the terminal:

  ```
  anaconda-navigator
  ```

Step 5: Set Up Environments and Install Packages

1. **Create a New Environment:**
    - To create a new environment with a specific version of Python, use:

      ```
      conda create --name myenv python=3.x
      ```

      Replace `myenv` with your environment name and `3.x` with the Python version you need.

2. **Activate the Environment:**
    - Activate yor environment with:

      ```
      conda activate myenv
      ```

3. **Install Packages:**
    - Install packages using Conda or pip:

      ```
      conda install package_name
      ```

      or

      ```
      pip install package_name
      ```

Step 6: Update Anaconda (Optional)

To keep Anaconda up to date, use the following command:

```
conda update conda
```

You can also update all packages using:

```
conda update --all
```

Using Anaconda for Python Development

Once Anaconda is installed, you can start using it for Python development. You can use **Jupyter Notebook**, **Spyder**, or any other IDE that comes with Anaconda. Additionally, you can manage packages and environments easily with `conda` commands, making Anaconda a powerful tool for data science, machine learning, and more.

Installing and Using Jupyter Notebook:

Jupyter Notebook is an open-source web application that allows you to create and share documents containing live code, equations, visualizations, and narrative text. It is widely used in data science, machine learning, and scientific computing for interactive programming and data analysis. Below is a step-by-step guide on how to install and use Jupyter Notebook.

Step 1: Install Jupyter Notebook

There are several ways to install Jupyter Notebook, but the easiest method is to use the Anaconda distribution, which comes with Jupyter pre-installed. If you haven't installed Anaconda yet, follow the steps in the previous section to install it.

Option 1: Using Anaconda (Recommended)

1. **Open Anaconda Navigator**: If you have Anaconda installed, you can launch Jupyter Notebook directly from the Anaconda Navigator.
    - Open Anaconda Navigator from the Start Menu (Windows) or Applications folder (macOS).
    - In the Navigator, click on "Jupyter Notebook" under the "Home" tab, then click "Launch."

2. **Install via Terminal/Command Prompt (if not already installed)**:
    - Open your terminal (macOS/Linux) or Anaconda Prompt (Windows).
    - Run the following command to ensure Jupyter Notebook is installed:

      ```
      conda install jupyter
      ```

    - This will install Jupyter Notebook along with all necessary dependencies.

Option 2: Using pip If you don’t have Anaconda installed, you can install Jupyter Notebook using pip:

1. **Install Jupyter**:
    - Open your terminal or command prompt.
    - Run the following command:

      ```
      pip install notebook
      ```

    - This will install Jupyter Notebook along with its dependencies.

Step 2: Launch Jupyter Notebook

Once installed, you can launch Jupyter Notebook from the terminal or Anaconda Navigator.

Option 1: Using Anaconda Navigator

1. **Open Anaconda Navigator**.
2. **Click on "Jupyter Notebook"** under the "Home" tab.
3. **Click "Launch"**.

Option 2: Using Terminal/Command Prompt

1. **Open Terminal/Command Prompt**.
2. **Run the following command**:

   ```
   jupyter notebook
   ```

   - This will start the Jupyter Notebook server and open a new tab in your default web browser.

3. **Jupyter Dashboard**:
   - The Jupyter Notebook dashboard will open in your web browser, showing the contents of the current working directory.
   - From here, you can create new notebooks, open existing ones, or organize your files.

Step 3: Create and Use a Jupyter Notebook

1. **Create a New Notebook**:
   - In the Jupyter Notebook dashboard, click the "New" button on the top right corner.
   - Select "Python 3" or another kernel you have installed.
   - This will create a new notebook and open it in a new tab.

2. **Using the Notebook Interface**:
   - **Cells**: Jupyter Notebook consists of cells. You can write code in code cells and markdown in markdown cells.
   - **Execute Code**: To run code, write your Python code in a code cell and press `Shift + Enter`. The output will appear directly below the cell.
   - **Markdown**: You can add markdown cells for notes, headings, or explanations. Use `Ctrl + M` to convert a cell to markdown, and `Shift + Enter` to render it.

3. **Add, Remove, and Modify Cells**:
   - **Add Cell**: Use the "Insert" menu or click the "+" button in the toolbar to add a new cell.
   - **Remove Cell**: Select a cell and click the scissors icon to remove it.
   - **Move Cells**: You can move cells up or down using the arrow buttons in the toolbar.

4. **Save and Export Notebooks**:
   - **Save**: Click the disk icon or use `Ctrl + S` (Windows/Linux) or `cmd + S` (macOS) to save your notebook.
   - **Export**: You can export your notebook to various formats, including PDF, HTML, and LaTeX, by going to `File -> Download as`.

Step 4: Installing and Managing Jupyter Extensions (Optional)

Jupyter Notebook supports a variety of extensions that can enhance its functionality. You can install and manage these extensions using `nbextensions`.

1. **Install Jupyter nbextensions**:
   - Open your terminal or command prompt and run:

     ```
     conda install -c conda-forge jupyter_contrib_nbextensions
     ```

   - This will install the nbextensions package and allow you to manage extensions through the Jupyter interface.

2. **Enable/Disable Extensions**:
   - After installation, launch Jupyter Notebook.
   - Go to the "Nbextensions" tab in the Jupyter dashboard to browse and enable/disable extensions.
   - Extensions like "Table of Contents" or "Codefolding" can improve your workflow.

Step 5: Running Jupyter Notebook on a Remote Server (Optional)

Jupyter Notebook can also be run on a remote server, allowing you to access it via a web browser from another device.

1. **SSH into Remote Server**:
   - Connect to your remote server using SSH:

     ```
     ssh username@remote_server_ip
     ```

2. **Launch Jupyter Notebook**:
   - On the remote server, run the following command to start Jupyter Notebook:

     ```
     jupyter notebook --no-browser --port=8888
     ```

   - This will start Jupyter Notebook on the server without opening a browser.

3. **Create an SSH Tunnel**:
   - On your local machine, create an SSH tunnel to the remote server:

     ```
     ssh -L 8888:localhost:8888 username@remote_server_ip
     ```

   - Replace `8888` with the port number you used when starting Jupyter Notebook.

4. **Access Jupyter Notebook**:
   - Open your web browser and go to `http://localhost:8888`.
   - You should now see the Jupyter Notebook interface running on the remote server.

Step 6: Using Jupyter Notebook for Data Science and Visualization

Jupyter Notebook is particularly powerful for data science and visualization tasks.

1. **Install Data Science Libraries**:
   - Use `conda` or `pip` to install libraries like `pandas`, `numpy`, `matplotlib`, and `seaborn`:

     ```
     conda install pandas numpy matplotlib seaborn
     ```

2. **Data Analysis**:
   - Use `pandas` for data manipulation, `numpy` for numerical operations, and `matplotlib` or `seaborn` for data visualization.

3. **Interactive Widgets**:
   - You can install and use interactive widgets in your notebook using the `ipywidgets` library:

     ```
     conda install ipywidgets
     ```

4. **Document and Share**:
   - Use markdown cells to document your work, and export notebooks as HTML or PDF to share your findings.

Step 7: Shutting Down Jupyter Notebook

1. **Shut Down Notebooks**:
   - To shut down a notebook, go back to the Jupyter Notebook dashboard, find the running notebook under the "Running" tab, and click "Shutdown."

2. **Stop the Jupyter Notebook Server**:
   - To stop the server, go to the terminal where you started Jupyter Notebook and press `Ctrl + C`. Confirm with `y` to stop the server.

Parts of Python Programming Language:

Identifiers:-

Identifiers are names used to identify variables, functions, classes, modules, and other objects in Python. They must follow specific rules:

1. **Naming Rules**:
   - **Must Start with a Letter or Underscore**: An identifier must begin with a letter (a-z, A-Z) or an underscore (_).
   - **Can Include Digits**: After the first character, digits (0-9) are allowed.

- **Case-Sensitive**: Identifiers are case-sensitive, meaning `variable`, `Variable`, and `VARIABLE` are considered different identifiers.
- **No Special Characters**: Identifiers cannot include special characters like `@`, `$`, or `-`.

2. **Examples**:
   - Valid Identifiers: `my_variable`, `_temp`, `value1`, `calculate_sum`
   - Invalid Identifiers: `2value` (starts with a digit), `my-variable` (contains a hyphen), `class` (a reserved keyword)

Rules for Naming Identifiers

1. **Start with a Letter or Underscore**:
   - An identifier must start with a letter (a-z, A-Z) or an underscore (_).
   - Examples:
     - Valid: `name`, `_value`, `first_name`
     - Invalid: `1name`, `@data`

2. **Followed by Letters, Digits, or Underscores**:
   - After the first character, the identifier can contain letters, digits (0-9), or underscores.
   - Examples:
     - Valid: `var1`, `count_2`, `data_set`
     - Invalid: `data-set`, `count%`

3. **Case-Sensitive**:
   - Python is case-sensitive, so `myVariable`, `MyVariable`, and `myvariable` are considered different identifiers.
   - Example:
     - `Total`, `total`, and `TOTAL` are different variables.

4. **No Reserved Keywords**:
   - Identifiers cannot be Python reserved keywords like `if`, `else`, `while`, `class`, etc.
   - Example:
     - Invalid: `if`, `try`, `class`
     - Valid: `if_var`, `class_name`

5. **Unlimited Length**:
   - Identifiers can be of any length in Python, but it's recommended to keep them concise and meaningful.

Best Practices for Naming Identifiers

1. **Descriptive Names**:
    - Use descriptive names for variables, functions, and classes that convey their purpose.
    - Example:
        - Instead of `x`, use `student_name` for clarity.

2. **Use of Underscores**:
    - For multi-word identifiers, use underscores to separate words (snake_case).
    - Example:
        - `user_name`, `total_count`, `max_value`

3. **Avoid Single Characters**:
    - Avoid using single-character identifiers unless in loop counters (like `i`, `j`) or for temporary values.

4. **Constants in Uppercase**:
    - Conventionally, constants are written in all uppercase letters with underscores separating words.
    - Example:
        - `PI = 3.14`, `MAX_LIMIT = 1000`

5. **Avoid Confusing Names**:
    - Avoid names that are easily confused with each other, like `O` (capital letter O) and `0` (zero), or `l` (lowercase L) and `1` (one).

Examples of Valid and Invalid Identifiers

- **Valid Identifiers**:
    - `age`
    - `first_name`
    - `studentID`
    - `_private_var`
    - `data123`

- **Invalid Identifiers**:
    - `123name` (starts with a digit)
    - `first-name` (contains a hyphen)
    - `class` (reserved keyword)
    - `total%` (contains a special character `%`)

Identifiers in Different Contexts

- **Variables**:
    - `age = 25` (Here, `age` is an identifier representing a variable.)

- **Functions**:

- `def calculate_sum(a, b):` (Here, `calculate_sum` is an identifier representing a function.)

- **Classes**:
  - `class Student:` (Here, `Student` is an identifier representing a class.)

- **Modules**:
  - `import math` (Here, `math` is an identifier representing a module.)

Keywords:-

Keywords in Python are reserved words that have a predefined meaning and purpose. These words form the core of Python's syntax and cannot be used as identifiers (i.e., names for variables, functions, classes, etc.). Each keyword is associated with a specific functionality in the Python programming language.

List of Python Keywords

As of Python 3.x, here is a list of all the keywords:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Explanation of Common Keywords

1. **`False` / `True`**:
   - Boolean values in Python representing truth values (similar to 0 and 1 in other languages).
   - Example:

     ```
     is_valid = True
     ```

2. **`None`**:
   - Represents the absence of a value or a null value.
   - Example:

     ```
     result = None
     ```

3. **`and`** / **`or`** / **`not`**:
    - Logical operators used in conditional statements.
    - Example:

```
if a > 0 and b > 0:
    print("Both are positive")
```

4. **`if`** / **`elif`** / **`else`**:
    - Used for conditional branching.
    - Example:

```
if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
else:
    grade = 'C'
```

5. **`for`** / **`while`**:
    - Used to create loops.
    - Example:

```
for i in range(5):
    print(i)
```

6. **`break`** / **`continue`**:
    - Control the flow of loops. `break` exits the loop, and `continue` skips the current iteration.
    - Example:

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

7. **`def`** / **`return`**:
    - Used to define a function and return a value from it.
    - Example:

```
def add(a, b):
    return a + b
```

8. **`class`**:
    - Used to define a class.
    - Example:

```
class Dog:
    pass
```

9. **`try` / `except` / `finally`**:
    - Used for exception handling.
    - Example:

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
finally:
    print("This will always execute")
```

10. **`import` / `from`**:
    - Used to import modules and specific elements from modules.
    - Example:

```
import math
from math import sqrt
```

11. **`lambda`**:
    - Used to create anonymous functions.
    - Example:

```
add = lambda x, y: x + y
```

12. **`with`**:
    - Used to simplify exception handling and automatically manage resources.
    - Example:

```
with open('file.txt', 'r') as file:
    content = file.read()
```

13. **`assert`**:
    - Used to test if a condition in your code returns `True`. If not, it raises an `AssertionError`.
    - Example:

```
assert x > 0, "x must be greater than 0"
```

14. **`yield`**:
    - Used to return a generator, which can be iterated over one value at a time.
    - Example:

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

15. **`async` / `await`**:

- Used to define asynchronous functions and to wait for asynchronous operations to complete.
- Example:

```
async def fetch_data():
    await some_async_function()
```

Reserved Nature of Keywords

- **Cannot be Used as Identifiers**: Since keywords have a special meaning in Python, they cannot be used as names for variables, functions, classes, or any other identifier. For example, using `if` as a variable name would result in a syntax error.
- **Case Sensitivity**: Python keywords are case-sensitive. For example, `True` is a keyword, but `true` is not.

How to Get the List of Keywords in Python

You can get the list of all Python keywords programmatically by using the `keyword` module:

```
import keyword
print(keyword.kwlist)
```

This will print a list of all the keywords available in your current Python version.

Statements:-

In Python, a statement is a unit of code that performs some action. Each statement in Python corresponds to a command or operation that the interpreter executes. Statements are the building blocks of Python programs, and they can range from simple assignments to complex control structures.

Types of Statements in Python

1. **Assignment Statements**
   - Used to assign values to variables.
   - Example:

```
x = 10
name = "Alice"
```

2. **Expression Statements**
   - An expression followed by a newline is treated as a statement.
   - Example:

```
5 + 3
```

3. **Print Statements**
    - Used to display output to the console.
    - Example:

```
print("Hello, world!")
```

4. **Control Flow Statements**
    - Used to control the execution flow of the program.
    - **Conditional Statements**:
        - `if`, `elif`, and `else` are used for branching based on conditions.
        - Example:

```
if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
else:
    print("Zero")
```

    - **Loop Statements**:
        - `for` and `while` loops are used to repeat code blocks.
        - Example (for loop):

```
for i in range(5):
    print(i)
```

        - Example (while loop):

```
i = 0
while i < 5:
    print(i)
    i += 1
```

    - **Control Statements in Loops**:
        - `break` and `continue` control the loop execution.
        - Example:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

5. **Function Definitions**
    - Defined using the `def` keyword.
    - Example:

```
def greet(name):
    return f"Hello, {name}"
```

6. **Class Definitions**
    - Defined using the `class` keyword.
    - Example:

```
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        return f"Hello, {self.name}"
```

7. **Import Statements**
    - Used to import modules or specific elements from modules.
    - Example:

```
import math
from datetime import datetime
```

8. **Exception Handling Statements**
    - Used to handle exceptions and errors.
    - **Try-Except Block**:
        - Example:

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

9. **With Statement**
    - Used for resource management, ensuring that resources are properly cleaned up after use.
    - Example:

```
with open('file.txt', 'r') as file:
    content = file.read()
```

10. **Assertion Statement**
    - Used for debugging purposes to test if a condition is true.
    - Example:

```
assert x > 0, "x should be positive"
```

11. **Return Statement**
    - Used to return a value from a function.
    - Example:

```
def add(a, b):
    return a + b
```

12. **Yield Statement**
    - Used to return a generator object from a function.
    - Example:

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1
```

Understanding Statement Execution

- **Sequential Execution**:
    - Statements are executed in the order they appear in the code, from top to bottom.

- **Indentation**:
    - Indentation is used to define blocks of code, especially in control structures, function definitions, and class definitions.

- **Line Continuation**:
    - A statement can be continued across multiple lines using a backslash (\) or by using parentheses, brackets, or braces.
    - Example:

```
total = (a + b +
         c)
```

Expressions:-

In Python, an expression is a combination of values, variables, operators, and function calls that are evaluated to produce a result. Expressions are used to perform operations and calculations, and they can be as simple as a single value or as complex as a series of operations and function calls.

Types of Expressions

1. **Literal Expressions**
    - Expressions that contain direct values.
    - Examples:

```
10          # Integer literal
3.14        # Floating-point literal
"Hello"     # String literal
```

2. **Arithmetic Expressions**

- Expressions that involve arithmetic operators to perform mathematical operations.
- Operators: `+`, `-`, `*`, `/`, `%`, `**`, `//`
- Examples:

```
x = 10 + 5          # Addition
y = 10 - 3          # Subtraction
z = 4 * 5           # Multiplication
a = 20 / 4          # Division
b = 20 % 3          # Modulus
c = 2 ** 3          # Exponentiation
d = 10 // 3         # Floor division
```

3. **Relational Expressions**
   - Expressions that compare values and return Boolean results (`True` or `False`).
   - Operators: `==`, `!=`, `>`, `<`, `>=`, `<=`
   - Examples:

```
a = (5 == 5)        # True
b = (10 > 3)        # True
c = (7 < 4)         # False
d = (5 != 5)        # False
```

4. **Logical Expressions**
   - Expressions that combine Boolean values using logical operators.
   - Operators: `and`, `or`, `not`
   - Examples:

```
x = (True and False)    # False
y = (True or False)     # True
z = not True            # False
```

5. **Bitwise Expressions**
   - Expressions that perform bit-level operations on integers.
   - Operators: `&`, `|`, `^`, `~`, `<<`, `>>`
   - Examples:

```
a = 5 & 3      # Bitwise AND
b = 5 | 3      # Bitwise OR
c = 5 ^ 3      # Bitwise XOR
d = ~5         # Bitwise NOT
e = 5 << 1     # Bitwise left shift
f = 5 >> 1     # Bitwise right shift
```

6. **Assignment Expressions**
   - Expressions that assign a value to a variable while evaluating to that value.
   - Introduced in Python 3.8 with the "walrus operator" (`:=`).
   - Example:

```
n = (x := 10) + 5    # x is assigned 10, and the expression
evaluates to 15
```

7. **Conditional Expressions**
    - Expressions that use conditional logic to select one of two values.
    - Syntax: `value_if_true if condition else value_if_false`
    - Example:

```
result = "Even" if x % 2 == 0 else "Odd"
```

8. **Function Calls**
    - Expressions that involve calling functions and using their return values.
    - Example:

```
result = len("Python")    # Function call expression
```

9. **Indexing and Slicing**
    - Expressions that access elements of sequences like lists, tuples, and strings.
    - Examples:

```
lst = [1, 2, 3, 4, 5]
item = lst[2]            # Indexing expression
sublist = lst[1:4]       # Slicing expression
```

10. **List Comprehensions**
    - Compact expressions for generating lists based on existing lists or iterables.
    - Example:

```
squares = [x**2 for x in range(10)]    # List comprehension
```

Evaluation of Expressions

- **Order of Operations**:
    - Python follows standard mathematical precedence rules. For example, multiplication and division have higher precedence than addition and subtraction.
    - Parentheses can be used to alter the default precedence and grouping.
- **Short-Circuit Evaluation**:
    - In logical expressions, Python uses short-circuit evaluation. For example, in `True or some_function()`, `some_function()` is not called because the result is already determined by `True`.

Examples of Expressions

1. **Simple Arithmetic Expression**:

```
result = 10 * (5 + 2)  # Evaluates to 70
```

2. **Relational and Logical Expression**:

```
is_valid = (age > 18) and (membership == "premium")  # Evaluates to
True or False
```

3. **List Comprehension**:

```
evens = [x for x in range(10) if x % 2 == 0]  # Evaluates to [0, 2, 4,
6, 8]
```

4. **Conditional Expression**:

```
status = "Adult" if age >= 18 else "Minor"  # Evaluates to "Adult" or
"Minor"
```

Variables:-

In Python, a variable is a symbolic name that refers to a value stored in memory. Variables are used to hold data that can be referenced and manipulated throughout your program. Unlike some other programming languages, Python does not require explicit declaration of variable types; instead, variables are dynamically typed and can be assigned to different types of data during runtime.

Characteristics of Variables in Python

1. **Dynamic Typing**
   - Variables in Python are dynamically typed, meaning you don't need to specify their type when you declare them. The type is inferred from the value assigned.
   - Example:

```
x = 10        # x is an integer
x = "Hello"   # Now x is a string
```

2. **No Explicit Declaration**
   - You don't need to declare a variable before using it. Simply assigning a value to a name creates a variable.
   - Example:

```
name = "Alice"
age = 30
```

3. **Case Sensitivity**
   - Variable names are case-sensitive. `age`, `Age`, and `AGE` are different variables.
   - Example:

```
age = 25
Age = 30
```

4. **Naming Conventions**
    - Variable names must start with a letter (a-z, A-Z) or an underscore (_).
    - The rest of the name can include letters, digits (0-9), and underscores.
    - Names are typically written in lowercase with words separated by underscores (snake_case).
    - Examples:

```
user_name = "John"
_temp = 20
max_value = 100
```

5. **Immutable vs Mutable Types**
    - **Immutable Types**: Variables that store immutable objects (e.g., integers, floats, strings, tuples). The value cannot be changed once created.
    - **Mutable Types**: Variables that store mutable objects (e.g., lists, dictionaries, sets). The value can be changed after creation.
    - Examples:

```
# Immutable
number = 10
number = 15   # Reassigning to a new integer

# Mutable
data = [1, 2, 3]
data.append(4)  # Modifying the list
```

6. **Global vs Local Variables**
    - **Global Variables**: Defined outside of functions and accessible throughout the module.
    - **Local Variables**: Defined inside functions and accessible only within that function.
    - Example:

```
global_var = "I'm global"

def my_function():
    local_var = "I'm local"
    print(global_var)  # Accessible
    print(local_var)   # Accessible

print(local_var)     # Error: local_var is not accessible here
```

7. **Rebinding Variables**
    - Variables can be rebound to different objects of any type at any time.
    - Example:

```
var = 10         # Initially an integer
var = [1, 2, 3]  # Now a list
```

8. **Multiple Assignments**

- You can assign the same value to multiple variables in one line or assign different values to multiple variables in one line.
- Examples:

```
x = y = z = 5            # All three variables are assigned the
same value
a, b, c = 1, 2, 3        # Multiple variables assigned different
values
```

Variable Scope and Lifetime

1. **Scope**:
   - **Local Scope**: Variables declared within a function or block are local to that function or block.
   - **Global Scope**: Variables declared outside of functions are global and accessible throughout the module.
   - **Nonlocal Scope**: Variables declared in nested functions can be referenced using the `nonlocal` keyword.

2. **Lifetime**:
   - Variables exist as long as they are in scope. Local variables are destroyed when the function exits, while global variables remain as long as the program runs.

Examples

1. **Basic Variable Assignment**:

```
name = "Alice"
age = 30
```

2. **Using Variables in Expressions**:

```
length = 5
width = 10
area = length * width
print("Area:", area)
```

3. **Changing Variable Types**:

```
x = 100          # x is an integer
x = "Python"     # Now x is a string
```

4. **Global and Local Variables**:

```
count = 0  # Global variable

def increment():
    global count
```

```
    count += 1

increment()
print(count)  # Output: 1
```

5. **Multiple Assignments**:

```
x, y, z = 1, 2, 3
print(x, y, z)  # Output: 1 2 3
```

Operators:-

Operators are symbols that perform operations on variables and values. Python supports a variety of operators, each designed to perform specific tasks. They can be classified into several categories:

1. Arithmetic Operators

Arithmetic operators perform basic mathematical operations.

- **Addition (+)**:

```
result = 5 + 3  # result is 8
```

- **Subtraction (-)**:

```
result = 5 - 3  # result is 2
```

- **Multiplication (*)**:

```
result = 5 * 3  # result is 15
```

- **Division (/)**:

```
result = 5 / 3  # result is approximately 1.6667
```

- **Floor Division (//)**:

```
result = 5 // 3  # result is 1
```

- **Modulus (%)**:

```
result = 5 % 3  # result is 2
```

- **Exponentiation (**)**:

```
result = 5 ** 3  # result is 125
```

2. Relational (Comparison) Operators

Relational operators compare values and return Boolean results (`True` or `False`).

- **Equal to (==)**:

  ```
  result = (5 == 3)  # result is False
  ```

- **Not equal to (!=)**:

  ```
  result = (5 != 3)  # result is True
  ```

- **Greater than (>)**:

  ```
  result = (5 > 3)  # result is True
  ```

- **Less than (<)**:

  ```
  result = (5 < 3)  # result is False
  ```

- **Greater than or equal to (>=)**:

  ```
  result = (5 >= 3)  # result is True
  ```

- **Less than or equal to (<=)**:

  ```
  result = (5 <= 3)  # result is False
  ```

3. Logical Operators

Logical operators are used to combine conditional statements.

- **Logical AND (and)**:

  ```
  result = (5 > 3) and (2 < 4)  # result is True
  ```

- **Logical OR (or)**:

  ```
  result = (5 > 3) or (2 > 4)  # result is True
  ```

- **Logical NOT (not)**:

  ```
  result = not (5 > 3)  # result is False
  ```

4. Bitwise Operators

Bitwise operators perform operations on binary representations of integers.

- **Bitwise AND (&)**:

```
result = 5 & 3  # result is 1 (binary 0101 & 0011 = 0001)
```

- **Bitwise OR (|)**:

```
result = 5 | 3  # result is 7 (binary 0101 | 0011 = 0111)
```

- **Bitwise XOR (^)**:

```
result = 5 ^ 3  # result is 6 (binary 0101 ^ 0011 = 0110)
```

- **Bitwise NOT (~)**:

```
result = ~5  # result is -6 (binary ~0101 = 1010 with sign bit)
```

- **Bitwise Left Shift (<<)**:

```
result = 5 << 1  # result is 10 (binary 0101 << 1 = 1010)
```

- **Bitwise Right Shift (>>)**:

```
result = 5 >> 1  # result is 2 (binary 0101 >> 1 = 0010)
```

5. Assignment Operators

Assignment operators are used to assign values to variables.

- **Simple Assignment (=)**:

```
x = 5
```

- **Addition Assignment (+=)**:

```
x += 3  # Equivalent to x = x + 3
```

- **Subtraction Assignment (-=)**:

```
x -= 2  # Equivalent to x = x - 2
```

- **Multiplication Assignment (*=)**:

```
x *= 4  # Equivalent to x = x * 4
```

- **Division Assignment (`/=`)**:

```
x /= 2  # Equivalent to x = x / 2
```

- **Floor Division Assignment (`//=`)**:

```
x //= 2  # Equivalent to x = x // 2
```

- **Modulus Assignment (`%=`)**:

```
x %= 3  # Equivalent to x = x % 3
```

- **Exponentiation Assignment (`**=`)**:

```
x **= 2  # Equivalent to x = x ** 2
```

- **Bitwise AND Assignment (`&=`)**:

```
x &= 3  # Equivalent to x = x & 3
```

- **Bitwise OR Assignment (`|=`)**:

```
x |= 3  # Equivalent to x = x | 3
```

- **Bitwise XOR Assignment (`^=`)**:

```
x ^= 3  # Equivalent to x = x ^ 3
```

- **Bitwise Left Shift Assignment (`<<=`)**:

```
x <<= 2  # Equivalent to x = x << 2
```

- **Bitwise Right Shift Assignment (`>>=`)**:

```
x >>= 2  # Equivalent to x = x >> 2
```

6. Membership Operators

Membership operators test for membership within sequences (lists, tuples, strings).

- **In (`in`)**:

```
result = 5 in [1, 2, 3, 5]  # result is True
```

- **Not In (`not in`)**:

```
result = 5 not in [1, 2, 3]  # result is True
```

7. Identity Operators

Identity operators compare the memory locations of two objects.

- **Is (`is`)**:

```
a = [1, 2, 3]
b = a
result = (a is b)  # result is True
```

- **Is Not (`is not`)**:

```
a = [1, 2, 3]
b = [1, 2, 3]
result = (a is not b)  # result is True
```

8. Conditional Expressions

Conditional expressions evaluate expressions based on conditions.

- **Ternary Conditional Operator**:

```
result = "Even" if x % 2 == 0 else "Odd"
```

Precedence and Associativity:-

In Python, **precedence** and **associativity** determine the order in which operators are evaluated in expressions. Understanding these concepts is crucial for writing correct and predictable code.

1. Operator Precedence

Operator precedence refers to the order in which operators are applied in expressions. Operators with higher precedence are evaluated before those with lower precedence. If multiple operators have the same precedence, associativity determines their order.

Here is a summary of Python's operator precedence from highest to lowest:

1. **Parentheses** (`()`)
2. **Exponentiation** (`**`)
3. **Unary Plus and Minus** (`+x`, `-x`), **Unary Not** (`not x`), **Bitwise NOT** (`~x`)
4. **Multiplication, Division, Floor Division, Modulus** (`*`, `/`, `//`, `%`)

5. **Addition and Subtraction** (`+`, `-`)
6. **Bitwise Shift Operators** (`<<`, `>>`)
7. **Bitwise AND** (`&`)
8. **Bitwise XOR** (`^`)
9. **Bitwise OR** (`|`)
10. **Comparison Operators** (`==`, `!=`, `>`, `<`, `>=`, `<=`)
11. **Logical AND** (`and`)
12. **Logical OR** (`or`)
13. **Conditional Expression** (`if-else`)
14. **Assignment Operators** (`=`, `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `&=`, `|=`, `^=`, `<<=`, `>>=`)
15. **Expressions involving lambda** (`lambda`)

Operator Precedence Table

Precedence Level	Operators
1	`()`
2	`**`
3	`+x`, `-x`, `not x`, `~x`
4	`*`, `/`, `//`, `%`
5	`+`, `-`
6	`<<`, `>>`
7	`&`
8	`^`
9	`` ` ``
10	`==`, `!=`, `>`, `<`, `>=`, `<=`
11	`and`
12	`or`
13	`if-else`
14	`=`, `+=`, `-=`, `*=`, `/=`, `//=`, `%=`
15	`lambda`

2. Associativity

Associativity determines the order in which operators of the same precedence level are evaluated.

- **Left-to-Right Associativity**: Most operators in Python (e.g., `+`, `-`, `*`, `/`) are evaluated from left to right.

```
result = 2 + 3 - 1  # Evaluated as (2 + 3) - 1
```

- **Right-to-Left Associativity**: Some operators, such as the exponentiation operator (`**`) and assignment operators (=, +=, etc.), are evaluated from right to left.

```
result = 2 ** 3 ** 2  # Evaluated as 2 ** (3 ** 2)
```

Examples

1. **Expression with Multiple Operators**:

```
result = 2 + 3 * 4  # Multiplication (*) has higher precedence than
addition (+)
# Evaluated as 2 + (3 * 4) => 2 + 12 => 14
```

2. **Parentheses to Alter Precedence**:

```
result = (2 + 3) * 4  # Parentheses alter precedence
# Evaluated as (2 + 3) * 4 => 5 * 4 => 20
```

3. **Exponentiation Associativity**:

```
result = 2 ** 3 ** 2  # Right-to-left associativity
# Evaluated as 2 ** (3 ** 2) => 2 ** 9 => 512
```

4. **Logical Operators**:

```
result = True or False and False  # Logical AND has higher precedence
than OR
# Evaluated as True or (False and False) => True or False => True
```

Data Types:-

Python supports a variety of data types, allowing you to store and manipulate different kinds of data. Understanding these data types is essential for writing effective Python programs.

1. Basic Data Types

1. **Integer (`int`)**
    - Represents whole numbers without a fractional part.
    - Example:

      ```
      num = 42
      ```

2. **Floating-Point (`float`)**
    - Represents real numbers with a decimal point.
    - Example:

      ```
      pi = 3.14159
      ```

3. **String (`str`)**
    - Represents a sequence of characters enclosed in quotes.
    - Strings can be enclosed in single quotes, double quotes, or triple quotes (for multi-line strings).
    - Example:

      ```
      message = "Hello, World!"
      ```

4. **Boolean (`bool`)**
    - Represents one of two values: `True` or `False`.
    - Example:

      ```
      is_active = True
      ```

5. **None Type (`NoneType`)**
    - Represents the absence of a value or a null value.
    - Example:

      ```
      result = None
      ```

2. Collection Data Types

1. **List (`list`)**
    - An ordered collection of items that can be of different types. Lists are mutable.
    - Example:

      ```
      numbers = [1, 2, 3, 4, 5]
      ```

2. **Tuple (`tuple`)**
    - An ordered collection of items that can be of different types. Tuples are immutable.
    - Example:

      ```
      point = (2, 3)
      ```

3. **Set (`set`)**
    - An unordered collection of unique items. Sets are mutable.
    - Example:

```
unique_numbers = {1, 2, 3, 4, 5}
```

4. **Dictionary (`dict`)**
    - A collection of key-value pairs. Keys must be unique and immutable, while values can be of any type. Dictionaries are mutable.
    - Example:

```
student = {'name': 'Alice', 'age': 20}
```

3. Specialized Data Types

1. **Complex (`complex`)**
    - Represents complex numbers with a real and imaginary part.
    - Example:

```
z = 3 + 4j
```

2. **Bytes (`bytes`)**
    - Immutable sequence of bytes, often used for binary data.
    - Example:

```
byte_data = b'hello'
```

3. **Bytearray (`bytearray`)**
    - Mutable sequence of bytes.
    - Example:

```
byte_array = bytearray([65, 66, 67])
```

4. **Memoryview (`memoryview`)**
    - A view object that exposes an array's buffer interface.
    - Example:

```
data = bytearray(b'hello')
view = memoryview(data)
```

Note:-

1. Numeric Types: 1) int 2) float 3) complex
2. Sequence Types: 1) String 2) list 3) Tuple 4) Range
3. Set Type: 1) set 2) fronzenset (Frozen Set)
4. Mapping Type: dict (Dictionary)

5. Boolean Type: Boolean (`bool`) represents `True` or `False` values.
6. None Type**:** None (`NoneType`) represents the absence of a value or a null value.

Example:

```
x = 10  # Integer

y = 3.14  # Float

z = 1 + 2j  # Complex

name = "Alice"  # String

numbers = [1, 2, 3]  # List

coordinates = (10, 20)  # Tuple

person = {"name": "Alice", "age": 30}  # Dictionary

unique_numbers = {1, 2, 3}  # Set

immutable_set = frozenset([1, 2, 3])  # Frozen Set

is_active = True  # Boolean

result = None  # None
```

Indentation:-

Indentation is a fundamental aspect of Python's syntax. Unlike many other programming languages that use braces or keywords to define blocks of code, Python uses indentation to delimit blocks. Proper indentation is crucial for writing syntactically correct and functional Python code.

1. Importance of Indentation

- **Defines Code Blocks**: In Python, indentation defines the boundaries of code blocks such as loops, conditionals, and function definitions.
- **Syntax Requirement**: Python enforces indentation strictly. Improper indentation results in `IndentationError` or `SyntaxError`.

2. Indentation Rules

- **Consistency**: Use either spaces or tabs for indentation, but not both. The Python community recommends using spaces (typically 4 spaces per indentation level) for consistency.
- **Level of Indentation**: Each block of code following a colon (`:`) must be indented. The standard practice is to use 4 spaces per indentation level.

3. Examples

1. **Basic Example**:

```
if True:
    print("This is indented")
```

2. **Function Definition**:

```
def greet(name):
    if name:
        print(f"Hello, {name}!")
    else:
        print("Hello, World!")
```

3. **Loop Structures**:

```
for i in range(3):
    print(f"Iteration {i}")
    if i == 1:
        print("Special case")
```

4. **Nested Blocks**:

```
def process_list(lst):
    for item in lst:
        if item % 2 == 0:
            print(f"Even: {item}")
        else:
            print(f"Odd: {item}")
```

4. Indentation Errors

- **`IndentationError`**: Raised when the indentation is not consistent or incorrect.

```
if True:
print("This will raise an error")
```

  - **Error Message**: `IndentationError: expected an indented block`

- **`SyntaxError`**: May also occur if the indentation is inconsistent.

```
def example():
    print("Start")
  print("This will cause a SyntaxError")
```

  - **Error Message**: `SyntaxError: unindent does not match any outer indentation level`

5. Best Practices

- **Use Spaces**: Follow the convention of using 4 spaces per indentation level.
- **Configure Editors**: Most code editors and IDEs can be configured to automatically use spaces for indentation and ensure consistent indentation levels.
- **Be Consistent**: Maintain a consistent style throughout your codebase to enhance readability and maintainability.

6. Tools and Configuration

- **Editor Settings**: Configure your text editor or IDE to use 4 spaces for indentation. For example, in VS Code, you can set this in your settings.
- **Linting Tools**: Use tools like `flake8` or `pylint` to check for indentation issues and ensure code consistency.

Comments:-

Comments are an essential part of programming as they provide explanatory notes within the code. They help others (and yourself) understand the purpose and functionality of the code. Python supports different types of comments, each serving specific needs.

1. Types of Comments

1.1. Single-Line Comments

- **Syntax**: Use the # symbol to denote a single-line comment.
- **Usage**: Ideal for brief explanations or annotations on a single line.
- **Example**:

```
# This is a single-line comment
x = 10  # Initialize x with 10
```

1.2. Multi-Line Comments

- **Syntax**: Use triple quotes (`'''` or `"""`) for multi-line comments, though they are technically multi-line string literals.
- **Usage**: Suitable for longer comments spanning multiple lines. It's common practice to use them for docstrings (more on this below).
- **Example**:

```
"""
This is a multi-line comment.
It spans multiple lines and can be used to provide detailed
explanations.
"""
y = 20
```

2. Docstrings

- **Purpose**: Docstrings are a special type of multi-line comment used to document modules, classes, functions, and methods.
- **Syntax**: Enclosed in triple quotes (`'''` or `"""`), placed immediately after the function or class definition.
- **Usage**: Provides documentation for code and can be accessed programmatically via the `__doc__` attribute.
- **Example**:

```
def add(a, b):
    """
    Add two numbers and return the result.

    Parameters:
    a (int): The first number.
    b (int): The second number.

    Returns:
    int: The sum of a and b.
    """
    return a + b

print(add.__doc__)  # Output the docstring
```

3. Best Practices for Comments

1. **Be Clear and Concise**:
   - Write comments that are easy to understand and directly related to the code they annotate.

2. **Avoid Redundancy**:
   - Avoid stating the obvious. For instance, `# Increment i by 1` is redundant if `i += 1` is already clear.

3. **Keep Comments Up-to-Date**:
   - Update comments when the code changes to prevent discrepancies between the code and its documentation.

4. **Use Comments for Explanation, Not Just Annotation**:
   - Use comments to explain why certain decisions are made, not just what the code does.

5. **Document Complex Logic**:
   - Provide comments for complex or non-obvious sections of code to enhance understanding.

4. Examples

- **Single-Line Comment**:

```
# Calculate the area of a circle
radius = 5
area = 3.14 * radius * radius
```

- **Multi-Line Comment**:

```
"""
The following block of code calculates the factorial of a number.
Factorial is defined as the product of all positive integers up to the
number.
"""
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

- **Docstring Example**:

```
class Rectangle:
    """
    A class to represent a rectangle.

    Attributes:
    width (float): The width of the rectangle.
    height (float): The height of the rectangle.
    """

    def __init__(self, width, height):
        """
        Initialize a rectangle with width and height.

        Parameters:
        width (float): The width of the rectangle.
        height (float): The height of the rectangle.
        """
        self.width = width
        self.height = height
```

Reading Input:-

Reading input in Python is a common task that allows you to interact with users and obtain data from various sources. The primary method for reading user input is through the `input()` function. For more advanced use cases, Python provides additional libraries and methods.

1. Using the `input()` Function

The `input()` function reads a line of text from the user and returns it as a string. You can provide a prompt message to guide the user.

Syntax

```
input(prompt)
```

- **`prompt`**: Optional. A string that is displayed to the user before reading the input.

Examples

1. **Basic Input Example**:

   ```
   name = input("Enter your name: ")
   print(f"Hello, {name}!")
   ```

2. **Reading and Using Numerical Input**:
   - By default, `input()` returns data as a string. To work with numbers, you need to convert the input using type conversion functions.

   ```
   age = input("Enter your age: ")
   age = int(age)  # Convert to integer
   print(f"You are {age} years old.")
   ```

3. **Handling Multiple Inputs**:
   - You can read multiple values in a single line and split them into separate variables.

   ```
   x, y = input("Enter two numbers separated by space: ").split()
   x = int(x)
   y = int(y)
   print(f"Sum: {x + y}")
   ```

4. **Input Validation**:
   - To ensure the input is valid, you can use a loop to keep prompting the user until valid input is provided.

   ```
   while True:
       try:
           number = int(input("Enter a number: "))
           break
       except ValueError:
           print("That's not a valid number. Please try again.")
   print(f"You entered {number}.")
   ```

2. Reading Input from Files

For reading data from files, you can use Python's built-in file handling methods.

Syntax

```
with open(filename, mode) as file:
    content = file.read()
```

- **`filename`**: The name of the file to open.
- **`mode`**: The mode in which to open the file (e.g., `'r'` for read, `'w'` for write).

Examples

1. **Reading Entire File**:

   ```
   with open('example.txt', 'r') as file:
       content = file.read()
       print(content)
   ```

2. **Reading Line by Line**:

   ```
   with open('example.txt', 'r') as file:
       for line in file:
           print(line.strip())  # .strip() removes leading/trailing
   whitespace
   ```

3. **Reading Lines into a List**:

   ```
   with open('example.txt', 'r') as file:
       lines = file.readlines()
       for line in lines:
           print(line.strip())
   ```

3. Reading Input from the Command Line

Command line arguments allow you to pass data to a Python script when it is executed from the command line. This is useful for making scripts more flexible and configurable. Python provides several methods for handling command line arguments, with the `argparse` module being the most powerful and commonly used tool.

3.1 Using `sys.argv`

The `sys` module provides access to command line arguments via the `sys.argv` list. This method is simple but less flexible compared to `argparse`.

Syntax

```
import sys
```

```
# sys.argv is a list of command line arguments
# sys.argv[0] is the script name
# sys.argv[1:] are the arguments passed to the script
```

Examples

1. **Basic Example**:

   ```
   import sys

   # Print all command line arguments
   print("Script name:", sys.argv[0])
   print("Arguments:", sys.argv[1:])
   ```

   - **Running the script**:

     ```
     python script.py arg1 arg2
     ```

   - **Output**:

     ```
     Script name: script.py
     Arguments: ['arg1', 'arg2']
     ```

2. **Handling Arguments**:

   ```
   import sys

   if len(sys.argv) != 3:
       print("Usage: python script.py <arg1> <arg2>")
       sys.exit(1)

   arg1 = sys.argv[1]
   arg2 = sys.argv[2]
   print(f"Argument 1: {arg1}")
   print(f"Argument 2: {arg2}")
   ```

3.2 Using `argparse`

The `argparse` module provides a more robust and flexible way to handle command line arguments. It supports type conversion, argument validation, and generates help messages.

Syntax

```
import argparse

parser = argparse.ArgumentParser(description="Description of your program.")
parser.add_argument('argument_name', type=data_type, help='Description of
argument')
# Optionally add more arguments
args = parser.parse_args()
```

Examples

1. **Basic Usage**:

```
import argparse

parser = argparse.ArgumentParser(description="Process some integers.")
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
args = parser.parse_args()

print(f"Arguments: {args.integers}")
```

   - **Running the script**:

```
python script.py 1 2 3
```

   - **Output**:

```
Arguments: [1, 2, 3]
```

2. **Optional Arguments**:

```
import argparse

parser = argparse.ArgumentParser(description="Process some integers.")
parser.add_argument('-v', '--verbose', action='store_true',
                    help='increase output verbosity')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
args = parser.parse_args()

if args.verbose:
    print("Verbosity turned on")
print(f"Arguments: {args.integers}")
```

   - **Running the script**:

```
python script.py 1 2 3 --verbose
```

   - **Output**:

```
Verbosity turned on
Arguments: [1, 2, 3]
```

3. **Specifying Data Types and Default Values**:

```
import argparse

parser = argparse.ArgumentParser(description="Process some numbers.")
parser.add_argument('--number', type=int, default=10,
```

```
                    help='an integer number (default: 10)')
args = parser.parse_args()

print(f"Number: {args.number}")
```

- **Running the script**:

  ```
  python script.py --number 42
  ```

- **Output**:

  ```
  Number: 42
  ```

3.3 Using `click` Library

For more advanced command line interfaces, the `click` library provides additional features and a more user-friendly API.

Installation

```
pip install click
```

Examples

1. **Basic Usage**:

   ```
   import click

   @click.command()
   @click.argument('name')
   def greet(name):
       click.echo(f"Hello, {name}!")

   if __name__ == '__main__':
       greet()
   ```

   - **Running the script**:

     ```
     python script.py Alice
     ```

   - **Output**:

     ```
     Hello, Alice!
     ```

Printing Output:-

Printing output in Python is a fundamental task used to display information to the user. The `print()` function is the primary method for producing output in Python. It allows you to print strings, numbers, and other data types to the console. You can also format output for more complex requirements.

1. Basic Usage of `print()`

Syntax

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

- **`*objects`**: One or more objects to be printed.
- **`sep`**: Separator between objects (default is a space).
- **`end`**: String appended after the last object (default is a newline).
- **`file`**: A file-like object where the output is written (default is `sys.stdout`).
- **`flush`**: Whether to forcibly flush the stream (default is `False`).

Examples

1. **Printing a Simple Message**:

   ```
   print("Hello, World!")
   ```

   - **Output**:

     ```
     Hello, World!
     ```

2. **Printing Multiple Items**:

   ```
   name = "Alice"
   age = 30
   print("Name:", name, "Age:", age)
   ```

   - **Output**:

     ```
     Name: Alice Age: 30
     ```

3. **Custom Separator**:

   ```
   print("Python", "is", "fun", sep="-")
   ```

   - **Output**:

     ```
     Python-is-fun
     ```

4. **Custom End Character**:

   ```
   print("Hello", end=" ")
   ```

```
print("World!")
```

- **Output**:

```
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 `output.txt`.

6. **Flushing Output**:

```
import time
print("Loading", end="", flush=True)
time.sleep(1)
print(".", end="", flush=True)
time.sleep(1)
print(".", end="", flush=True)
```

- This will display a loading message with dots appearing one by one.

---

2. String Formatting

Python provides several ways to format strings for more control over the output.

2.1. Using `str.format()` Method

- **Syntax**:

```
"format string".format(values)
```

- **Examples**:

```
name = "Bob"
age = 25
print("Name: {}, Age: {}".format(name, age))
```

- **Output**:

```
Name: Bob, Age: 25
```

2.2. Using f-Strings (Python 3.6+)

- **Syntax**:

```
f"string {variable}"
```

- **Examples**:

```
name = "Carol"
age = 28
print(f"Name: {name}, Age: {age}")
```

  - **Output**:

```
Name: Carol, Age: 28
```

2.3. Using % Formatting

- **Syntax**:

```
"format string % values"
```

- **Examples**:

```
name = "David"
age = 35
print("Name: %s, Age: %d" % (name, age))
```

  - **Output**:

```
Name: David, Age: 35
```

3. Printing Data Structures

1. **Lists**:

```
fruits = ["apple", "banana", "cherry"]
print(fruits)
```

   - **Output**:

```
['apple', 'banana', 'cherry']
```

2. **Dictionaries**:

```
student = {"name": "Eve", "age": 22}
print(student)
```

   - **Output**:

```
{'name': 'Eve', 'age': 22}
```

3. **Formatted Printing of Lists**:

```
fruits = ["apple", "banana", "cherry"]
print("Fruits: ", ", ".join(fruits))
```

   - **Output**:

```
Fruits: apple, banana, cherry
```

4. Handling Special Characters

1. **Newline (\n)**:

```
print("Line 1\nLine 2")
```

   - **Output**:

```
Line 1
Line 2
```

2. **Tab (\t)**:

```
print("Name:\tAlice\nAge:\t30")
```

   - **Output**:

```
Name:   Alice
Age:    30
```

3. **Backslash (\\)**:

```
print("This is a backslash: \\")
```

   - **Output**:

```
This is a backslash: \
```

Type Conversions:-

Type conversion, or type casting, refers to the process of converting a value of one data type to another. Python provides several built-in functions to facilitate these conversions, making it easier to work with different types of data.

1. Implicit Type Conversion

Implicit type conversion, also known as coercion, happens automatically when Python performs operations involving different data types. The interpreter converts one data type to another to ensure compatibility.

Examples

1. **Integer to Float**:

```
x = 5        # Integer
y = 2.0      # Float
result = x + y  # Integer is implicitly converted to float
print(result)   # Output: 7.0
```

2. **Float to Complex**:

```
a = 3.5       # Float
b = 2 + 1j    # Complex
result = a + b  # Float is implicitly converted to complex
print(result)   # Output: (5.5+1j)
```

2. Explicit Type Conversion

Explicit type conversion requires the use of built-in functions to convert data from one type to another.

2.1. Converting to Integer

- **Syntax**:

```
int(x)
```

- **Examples**:
    1. **From Float**:

```
num = 3.14
integer_num = int(num)  # Converts float to integer
print(integer_num)     # Output: 3
```

    2. **From String**:

```
str_num = "123"
integer_num = int(str_num)  # Converts string to integer
print(integer_num)          # Output: 123
```

    3. **Invalid Conversion**:

```
str_num = "abc"
integer_num = int(str_num)  # Raises ValueError
```

2.2. Converting to Float

- **Syntax**:

  ```
  float(x)
  ```

- **Examples**:
  1. **From Integer**:

     ```
     num = 10
     float_num = float(num)  # Converts integer to float
     print(float_num)        # Output: 10.0
     ```

  2. **From String**:

     ```
     str_num = "3.14"
     float_num = float(str_num)  # Converts string to float
     print(float_num)            # Output: 3.14
     ```

  3. **Invalid Conversion**:

     ```
     str_num = "abc"
     float_num = float(str_num)  # Raises ValueError
     ```

2.3. Converting to String

- **Syntax**:

  ```
  str(x)
  ```

- **Examples**:
  1. **From Integer**:

     ```
     num = 456
     str_num = str(num)  # Converts integer to string
     print(str_num)      # Output: "456"
     ```

  2. **From Float**:

     ```
     num = 4.56
     str_num = str(num)  # Converts float to string
     print(str_num)      # Output: "4.56"
     ```

  3. **From List**:

```
lst = [1, 2, 3]
str_lst = str(lst)  # Converts list to string
print(str_lst)     # Output: "[1, 2, 3]"
```

2.4. Converting to List

- **Syntax**:

  ```
  list(iterable)
  ```

- **Examples**:
  1. **From Tuple**:

     ```
     tup = (1, 2, 3)
     lst = list(tup)  # Converts tuple to list
     print(lst)       # Output: [1, 2, 3]
     ```

  2. **From String**:

     ```
     str_data = "hello"
     lst = list(str_data)  # Converts string to list of characters
     print(lst)            # Output: ['h', 'e', 'l', 'l', 'o']
     ```

2.5. Converting to Tuple

- **Syntax**:

  ```
  tuple(iterable)
  ```

- **Examples**:
  1. **From List**:

     ```
     lst = [1, 2, 3]
     tup = tuple(lst)  # Converts list to tuple
     print(tup)        # Output: (1, 2, 3)
     ```

  2. **From String**:

     ```
     str_data = "hello"
     tup = tuple(str_data)  # Converts string to tuple of characters
     print(tup)            # Output: ('h', 'e', 'l', 'l', 'o')
     ```

3. Handling Conversion Errors

When performing type conversions, errors can occur if the conversion is invalid. It's good practice to handle these exceptions using `try` and `except` blocks.

Example

```
try:
    num = int("abc")  # Invalid conversion
except ValueError:
    print("Invalid input. Cannot convert to integer.")
```

- **Output**:

  ```
  Invalid input. Cannot convert to integer.
  ```

The `type()` Function and `is` Operator:-

In Python, the `type()` function and the `is` operator are used for type checking and comparison. Understanding their differences and use cases can help ensure your code behaves as expected.

---

1. The `type()` Function

The `type()` function is used to get the type of an object. It returns the type of the object passed to it as an argument.

Syntax

```
type(object)
```

Examples

1. **Basic Usage**:

   ```
   print(type(5))          # Output: <class 'int'>
   print(type(3.14))       # Output: <class 'float'>
   print(type("Hello"))   # Output: <class 'str'>
   ```

2. **Checking Types**:

   ```
   a = [1, 2, 3]
   if type(a) == list:
       print("a is a list")
   ```

   - **Output**:

     ```
     a is a list
     ```

3. **Determining the Type of a Custom Object**:

```
class MyClass:
    pass

obj = MyClass()
print(type(obj))  # Output: <class '__main__.MyClass'>
```

4. **Checking Type of Multiple Variables**:

```
x, y, z = 5, 3.14, "Hello"
print(type(x), type(y), type(z))
```

   - **Output**:

```
<class 'int'> <class 'float'> <class 'str'>
```

2. The `is` Operator

The `is` operator checks for object identity, i.e., whether two references point to the same object in memory. This is different from equality checking, which checks if two objects have the same value.

Syntax

```
object1 is object2
```

Examples

1. **Basic Usage**:

```
a = [1, 2, 3]
b = a
print(a is b)  # Output: True
```

2. **Comparing Different Objects**:

```
x = [1, 2, 3]
y = [1, 2, 3]
print(x is y)  # Output: False (x and y are different objects in
memory)
```

3. **Comparing with `None`**:

```
a = None
print(a is None)  # Output: True
```

4. **String Interning**:

```
a = "hello"
b = "hello"
print(a is b)  # Output: True (strings with the same value are
interned)
```

5. **Custom Objects**:

```
class MyClass:
    pass

obj1 = MyClass()
obj2 = MyClass()
print(obj1 is obj2)  # Output: False (obj1 and obj2 are different
instances)
```

3. `type()` vs. `is`

- **`type()`**: Used to check the type of an object. It provides information about what type of object you're dealing with (e.g., `int`, `str`, `list`).
- **`is`**: Used to check if two references point to the same object in memory. It is primarily used to compare identities and not values.

Examples of Usage

1. **Checking Object Type**:

```
num = 10
if type(num) is int:
    print("num is an integer")
```

   - **Output**:

```
num is an integer
```

2. **Checking Object Identity**:

```
list1 = [1, 2, 3]
list2 = list1
if list1 is list2:
    print("list1 and list2 are the same object")
```

   - **Output**:

```
list1 and list2 are the same object
```

Dynamic and Strongly Typed Languages:-

Python is often described as a dynamically typed and strongly typed language. Understanding these concepts can help you write more robust and predictable code.

1. Dynamic Typing

Dynamic typing means that the type of a variable is determined at runtime rather than at compile time. You do not need to declare the type of a variable when you create it; the type is inferred based on the value assigned to the variable.

Key Characteristics

- **Type Inference**: The type of a variable is inferred from the value assigned to it.
- **Flexibility**: You can reassign a variable to a different type of value.

Examples

1. **Type Inference**:

```
x = 10        # x is initially an integer
print(type(x))  # Output: <class 'int'>

x = "hello"    # x is now a string
print(type(x))  # Output: <class 'str'>
```

2. **Variable Reassignment**:

```
y = 3.14       # y is a float
print(type(y))  # Output: <class 'float'>

y = [1, 2, 3]  # y is now a list
print(type(y))  # Output: <class 'list'>
```

3. **Function Arguments**:

```
def add(a, b):
    return a + b

print(add(5, 3))    # Output: 8 (int + int)
print(add(5.0, 3.0))  # Output: 8.0 (float + float)
print(add("hello", " world"))  # Output: "hello world" (str + str)
```

2. Strongly Typed

Strongly typed means that Python enforces strict type rules, and you cannot perform operations on incompatible types without explicit conversion. Python will raise errors if you try to combine or operate on incompatible types.

Key Characteristics

- **Type Safety**: Operations on incompatible types will result in a `TypeError`.
- **Explicit Conversions**: You must explicitly convert between types when necessary.

Examples

1. **Type Error on Incompatible Types**:

```
num = 10
text = "Hello"
# Attempting to add an integer and a string
try:
    result = num + text
except TypeError as e:
    print(f"Error: {e}")  # Output: Error: unsupported operand type(s)
for +: 'int' and 'str'
```

2. **Explicit Type Conversion**:

```
num = 10
text = "20"
# Convert text to an integer before adding
result = num + int(text)
print(result)  # Output: 30
```

3. **Using `str()` for Explicit Conversion**:

```
number = 42
number_str = str(number)  # Convert integer to string
print(number_str)        # Output: "42"
```

4. **Combining Different Types**:

```
num = 5
text = "The number is"
combined = text + " " + str(num)  # Explicitly convert num to string
print(combined)  # Output: "The number is 5"
```

Comparison: Dynamic vs. Strongly Typed

- **Dynamic Typing**: Allows variables to change type at runtime and does not require type declarations.
- **Strongly Typed**: Enforces type rules and prevents operations on incompatible types without explicit conversions.

Example: Combining Both Concepts

```
def concatenate(value1, value2):
```

```
    return str(value1) + str(value2)

# Function uses dynamic typing (values can be of any type)
print(concatenate(5, 10))         # Output: "510"
print(concatenate("Hello", "World"))  # Output: "HelloWorld"
print(concatenate(5, " apples"))  # Output: "5 apples"
```

Conditional Statements / Control Flow Statements:-

Control statements in Python manage the flow of execution based on conditions and looping constructs. They include conditional statements and loop statements, which allow for decision-making and repetition in your programs.

1. Conditional Statements

Conditional statements enable you to execute certain blocks of code based on whether a condition evaluates to true or false.

1.1 `if` Statement

The `if` statement evaluates a condition and executes the associated block of code if the condition is true.

Syntax:

```
if condition:
    # code to execute if condition is true
```

Example:

```
age = 20
if age >= 18:
    print("You are an adult.")
```

- **Output:**

  ```
  sql
  Copy code
  You are an adult.
  ```

1.2 `if-else` Statement

The `if-else` statement executes one block of code if the condition is true and another block if it is false.

Syntax:

```
if condition:
    # code to execute if condition is true
else:
    # code to execute if condition is false
```

Example:

```
age = 16
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

- **Output:**

```
You are a minor.
```

1.3 `if...elif...else` Statement

The `if...elif...else` statement allows for multiple conditions to be checked in sequence.

Syntax:

```
if condition1:
    # code to execute if condition1 is true
elif condition2:
    # code to execute if condition2 is true
else:
    # code to execute if neither condition1 nor condition2 is true
```

Example:

```
score = 85
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: D")
```

- **Output:**

```
Grade: B
```

1.4 Nested `if` Statements

Nested `if` statements are used to check additional conditions within another `if` statement.

Syntax:

```
if condition1:
    if condition2:
        # code to execute if both conditions are true
    else:
        # code to execute if condition1 is true but condition2 is false
else:
    # code to execute if condition1 is false
```

Example:

```
age = 25
if age >= 18:
    if age < 21:
        print("You are an adult, but not old enough to drink.")
    else:
        print("You are an adult and can drink.")
else:
    print("You are a minor.")
```

- **Output:**

```
You are an adult and can drink.
```

2. Loop Statements

Loop statements allow you to repeat a block of code multiple times.

2.1 `while` Loop

The `while` loop continues to execute as long as a condition remains true.

Syntax:

```
while condition:
    # code to execute while condition is true
```

Example:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

- **Output:**

```
0
1
2
3
```

```
4
```

2.2 `for` Loop

The `for` loop iterates over a sequence (like a list, tuple, string, or range) and executes a block of code for each item.

Syntax:

```
for variable in sequence:
    # code to execute for each item in the sequence
```

Example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

- **Output:**

  ```
  apple
  banana
  cherry
  ```

- **Using `range()`**:

  ```
  for i in range(5):
      print(i)
  ```

  - **Output:**

    ```
    0
    1
    2
    3
    4
    ```

3. Control Flow Modifiers

3.1 `continue` Statement

The `continue` statement skips the rest of the code inside the loop for the current iteration and proceeds to the next iteration.

Syntax:

```
for item in sequence:
    if condition:
        continue
    # code to execute if condition is false
```

Example:

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

- **Output:**

```
0
1
3
4
```

3.2 `break` Statement

The `break` statement exits the loop prematurely, regardless of whether the loop's condition is still true.

Syntax:

```
for item in sequence:
    if condition:
        break
    # code to execute if condition is false
```

Example:

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

- **Output:**

```
0
1
2
```

Exception handling:-

Exception handling in Python is a mechanism to manage errors or unexpected events that occur during the execution of a program. By using exception handling, you can prevent your program from crashing and handle errors gracefully.

1. Introduction to Exceptions

An exception is an event that disrupts the normal flow of the program. It is usually caused by errors like division by zero, file not found, or invalid input. When an exception occurs, Python stops the current flow of the program and jumps to the nearest exception handler, if one exists.

2. Basic Syntax for Exception Handling

Python uses the `try`, `except`, `else`, and `finally` blocks to handle exceptions.

2.1 `try` and `except` Blocks

Syntax:

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
```

Example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

- **Output:**

  ```
  Cannot divide by zero.
  ```

2.2 `else` Block

The `else` block is executed if the code in the `try` block does not raise an exception.

Syntax:

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
else:
    # Code to execute if no exception occurs
```

Example:

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("Division successful. Result:", result)
```

- **Output:**

```
Division successful. Result: 5.0
```

2.3 `finally` Block

The `finally` block is always executed, regardless of whether an exception was raised or not. It is commonly used for cleanup actions.

Syntax:

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
finally:
    # Code to execute regardless of whether an exception occurs or not
```

Example:

```
try:
    file = open("test.txt", "r")
except FileNotFoundError:
    print("File not found.")
finally:
    file.close()
    print("File closed.")
```

- **Output:**

```
File not found.
File closed.
```

3. Raising Exceptions

You can raise exceptions manually using the `raise` statement. This is useful for creating custom error messages or when a specific condition needs to be handled as an error.

Syntax:

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

Example:

```
def divide(x, y):
    if y == 0:
        raise ValueError("Cannot divide by zero.")
    return x / y

try:
    result = divide(10, 0)
```

```
except ValueError as e:
    print(e)
```

- **Output:**

```
Cannot divide by zero.
```

4. Catching Multiple Exceptions

You can catch multiple exceptions by specifying multiple `except` blocks.

Syntax:

```
try:
    # Code that may raise an exception
except (ExceptionType1, ExceptionType2):
    # Code to handle the exceptions
```

Example:

```
try:
    value = int("abc")
except (ValueError, TypeError):
    print("ValueError or TypeError occurred.")
```

- **Output:**

```
ValueError or TypeError occurred.
```

5. Catching All Exceptions

You can catch all exceptions using a general `except` block. However, it is generally not recommended because it can make debugging difficult.

Syntax:

```
try:
    # Code that may raise an exception
except:
    # Code to handle any exception
```

Example:

```
try:
    result = 10 / 0
except:
    print("An unexpected error occurred.")
```

- **Output:**

```
An unexpected error occurred.
```

6. Custom Exceptions

You can define your own exception classes by inheriting from the built-in `Exception` class.

Syntax:

```
class CustomException(Exception):
    pass
```

Example:

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

try:
    raise MyError("This is a custom error message.")
except MyError as e:
    print(e)
```

- **Output:**

```
This is a custom error message.
```

-:Sample Experiments:-

Compute Area of circle: area = radius * radius * p

ComputeArea.py

```
'''
Compute Area of circle:
    area = radius * radius * p
'''
# Assign a value to radius
radius = 20 # radius is now 20
# Compute area
area = radius * radius * 3.14159
# Display results
print("The area for the circle of radius", radius, "is", area)
```

Output:

The area for the circle of radius 20 is 1256.636

Screenshot:

```
'''

Compute Area of circle:

    area = radius * radius * p

'''
# Assign a value to radius
radius = 20 # radius is now 20
# Compute area
area = radius * radius * 3.14159
# Display results
print("The area for the circle of radius", radius, "is", area)

```

```
input
The area for the circle of radius 20 is 1256.636
```

1. Write a program to find the largest element among three Numbers.

Ans:

Method-1: Using Conditional Statements:

```
# Input three numbers from the user

num1 = float(input("Enter the first number: "))

num2 = float(input("Enter the second number: "))

num3 = float(input("Enter the third number: "))

# Initialize the largest number

if num1 >= num2 and num1 >= num3:

    largest = num1

elif num2 >= num1 and num2 >= num3:

    largest = num2

else:

    largest = num3

print("The largest number is:", largest)
```

Method-2: Using the `max()` Function:-

```
# Input three numbers from the user

num1 = float(input("Enter the first number: "))

num2 = float(input("Enter the second number: "))

num3 = float(input("Enter the third number: "))

# Find the largest number using the max() function

largest = max(num1, num2, num3)


print("The largest number is:", largest)
```

Method-3: Using a List and `max()` Function:-

```
# Input three numbers from the user

num1 = float(input("Enter the first number: "))

num2 = float(input("Enter the second number: "))

num3 = float(input("Enter the third number: "))


# Store the numbers in a list

numbers = [num1, num2, num3]


# Find the largest number using the max() function

largest = max(numbers)


print("The largest number is:", largest)
```

Method-4:

Program: `find_largest.py`

```
# find_largest.py

def find_largest(num1, num2, num3):
    """
    Function to find the largest of three numbers.
    """
    if num1 >= num2 and num1 >= num3:
        return num1
    elif num2 >= num1 and num2 >= num3:
        return num2
    else:
        return num3

if __name__ == "__main__":
    # Input three numbers from the user
    try:
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))
        num3 = float(input("Enter the third number: "))

        # Find and print the largest number
        largest = find_largest(num1, num2, num3)
        print("The largest number is:", largest)
    except ValueError:
        print("Please enter valid numbers.")
```

How to Run the Program

1. **Save the Program**: Save the code above in a file named `find_largest.py`.
2. **Execute the Program**:
   - Open your command line or terminal.
   - Navigate to the directory where `find_largest.py` is saved.
   - Run the program using Python by typing:

     ```
     python find_largest.py
     ```

3. **Input Numbers**: When prompted, enter three numbers. The program will display the largest number among them.

Example Execution:

```
Enter the first number: 5
Enter the second number: 12
Enter the third number: 7
The largest number is: 12.0
```

2. Write a Program to display all prime numbers within an interval

Ans:

Program: `prime_numbers_interval_no_math.py`

```
# prime_numbers_interval_no_math.py

try:
    # Input the interval from the user
    start = int(input("Enter the start of the interval: "))
    end = int(input("Enter the end of the interval: "))

    if start > end:
        print("The start of the interval must be less than or equal to the
end.")
    else:
        print(f"Prime numbers between {start} and {end} are:")

        for number in range(start, end + 1):
            if number <= 1:
                continue
            if number == 2:
                print(number)
                continue
            if number % 2 == 0:
                continue

            # Determine the limit for checking divisibility
            limit = number // 2 + 1
            is_prime = True
            for i in range(3, limit, 2):
                if number % i == 0:
                    is_prime = False
                    break

            if is_prime:
                print(number)

except ValueError:
    print("Please enter valid integers.")
```

Run: python prime_numbers_interval_no_math.py

Output:-

Enter the start of the interval: 10

Enter the end of the interval: 30

Prime numbers between 10 and 30 are:

11

13

17

19

23

29

3. Write a program to swap two numbers without using a temporary variable.

Ans:

1. Tuple Unpacking

Python’s tuple unpacking feature allows you to swap two variables in a single line.

Example:

```
a = 5
b = 10

# Swapping using tuple unpacking
a, b = b, a

print(f"a = {a}, b = {b}")  # Output: a = 10, b = 5
```

2. Arithmetic Operations

This method uses addition and subtraction to swap values.

Example:

```
a = 5
b = 10

# Swapping using arithmetic operations
a = a + b
b = a - b
a = a - b

print(f"a = {a}, b = {b}")  # Output: a = 10, b = 5
```

3. Bitwise XOR Operation

The XOR bitwise operator can be used to swap values without a temporary variable.

Example:

```
a = 5
b = 10

# Swapping using XOR operation
a = a ^ b
b = a ^ b
a = a ^ b

print(f"a = {a}, b = {b}")  # Output: a = 10, b = 5
```

4. Using a Function Return

This approach involves defining a function that returns the swapped values.

Example:

```
def swap(a, b):
    return b, a

a = 5
b = 10

a, b = swap(a, b)

print(f"a = {a}, b = {b}")  # Output: a = 10, b = 5
```

5. Using Lists

This method involves using list indexing to swap values.

Example:

```
a = 5
b = 10

# Swapping using a list
lst = [a, b]
lst[0], lst[1] = lst[1], lst[0]
a, b = lst

print(f"a = {a}, b = {b}")  # Output: a = 10, b = 5
```

Example with functions:

Program: `swap_numbers_arithmetic.py`

```
# swap_numbers_arithmetic.py

def swap_numbers_arithmetic(a, b):
    """
    Swap two numbers using arithmetic operations.
```

```
    """
    print(f"Original values: a = {a}, b = {b}")

    # Swapping using arithmetic operations
    a = a + b
    b = a - b
    a = a - b

    print(f"Swapped values: a = {a}, b = {b}")

if __name__ == "__main__":
    try:
        # Input two numbers from the user
        a = float(input("Enter the first number: "))
        b = float(input("Enter the second number: "))

        # Swap the numbers
        swap_numbers_arithmetic(a, b)
    except ValueError:
        print("Please enter valid numbers.")
```

Note:

Swapping using tuple unpacking

a, b = b, a

4. Demonstrate the following Operators in Python with suitable examples.

i) Arithmetic Operators

ii) Relational Operators

iii) Assignment Operators

iv) Logical Operators

v) Bit wise Operators

vi) Ternary Operator

vii) Membership Operators

viii) Identity Operators

Ans:

i) Arithmetic Operators

Arithmetic operators are used to perform mathematical operations.

- **Addition (+)**: Adds two operands.
- **Subtraction (-)**: Subtracts the second operand from the first.
- **Multiplication (*)**: Multiplies two operands.
- **Division (/)**: Divides the first operand by the second (returns a float).
- **Integer Division (//)**: Divides the first operand by the second (returns an integer).
- **Modulus (%)**: Returns the remainder of the division.
- **Exponentiation (**)**: Raises the first operand to the power of the second.

Example:

```
a = 10
b = 3

print("Addition:", a + b)         # Output: 13
print("Subtraction:", a - b)      # Output: 7
print("Multiplication:", a * b)   # Output: 30
print("Division:", a / b)         # Output: 3.3333333333333335
print("Integer Division:", a // b) # Output: 3
print("Modulus:", a % b)          # Output: 1
print("Exponentiation:", a ** b)  # Output: 1000
```

ii) Relational Operators

Relational operators are used to compare two values.

- **Equal to (==)**: Checks if two values are equal.
- **Not equal to (!=)**: Checks if two values are not equal.
- **Greater than (>)**: Checks if the first value is greater than the second.
- **Less than (<)**: Checks if the first value is less than the second.
- **Greater than or equal to (>=)**: Checks if the first value is greater than or equal to the second.
- **Less than or equal to (<=)**: Checks if the first value is less than or equal to the second.

Example:

```
a = 10
b = 3

print("Equal to:", a == b)          # Output: False
print("Not equal to:", a != b)      # Output: True
print("Greater than:", a > b)       # Output: True
print("Less than:", a < b)          # Output: False
print("Greater than or equal to:", a >= b) # Output: True
print("Less than or equal to:", a <= b)    # Output: False
```

iii) Assignment Operators

Assignment operators are used to assign values to variables.

- **Assignment (=)**: Assigns a value to a variable.
- **Add and assign (+=)**: Adds and assigns a value.
- **Subtract and assign (-=)**: Subtracts and assigns a value.
- **Multiply and assign (*=)**: Multiplies and assigns a value.
- **Divide and assign (/=)**: Divides and assigns a value.
- **Integer divide and assign (//=)**: Integer divides and assigns a value.
- **Modulus and assign (%=)**: Takes modulus and assigns a value.
- **Exponentiate and assign (**=)**: Exponentiates and assigns a value.

Example:

```
a = 10
a += 5     # a = a + 5
print("Add and assign:", a)  # Output: 15

a -= 3     # a = a - 3
print("Subtract and assign:", a)  # Output: 12

a *= 2     # a = a * 2
print("Multiply and assign:", a)  # Output: 24

a /= 4     # a = a / 4
print("Divide and assign:", a)  # Output: 6.0
```

iv) Logical Operators

Logical operators are used to perform logical operations.

- **And (`and`)**: Returns `True` if both operands are `True`.
- **Or (`or`)**: Returns `True` if at least one operand is `True`.
- **Not (`not`)**: Returns `True` if the operand is `False`.

Example:

```
a = True
b = False

print("And:", a and b)    # Output: False
print("Or:", a or b)     # Output: True
print("Not a:", not a)   # Output: False
```

v) Bitwise Operators

Bitwise operators perform operations on the binary representations of integers.

- **And (`&`)**: Performs a bitwise AND operation.
- **Or (`|`)**: Performs a bitwise OR operation.

- **XOR (^)**: Performs a bitwise XOR operation.
- **Complement (~)**: Inverts the bits.
- **Left shift (<<)**: Shifts bits to the left.
- **Right shift (>>)**: Shifts bits to the right.

Example:

```
a = 5  # Binary: 0101
b = 3  # Binary: 0011

print("And:", a & b)         # Output: 1 (Binary: 0001)
print("Or:", a | b)          # Output: 7 (Binary: 0111)
print("XOR:", a ^ b)         # Output: 6 (Binary: 0110)
print("Complement:", ~a)    # Output: -6 (Binary: ...11111010)
print("Left shift:", a << 1) # Output: 10 (Binary: 1010)
print("Right shift:", a >> 1) # Output: 2 (Binary: 0010)
```

vi) Ternary Operator

The ternary operator is a shorthand for `if-else` statements.

Syntax:

```
value_if_true if condition else value_if_false
```

Example:

```
a = 10
b = 5

result = "a is greater" if a > b else "b is greater or equal"
print(result)  # Output: a is greater
```

vii) Membership Operators

Membership operators are used to test if a value is present in a sequence.

- **In (`in`)**: Returns `True` if the value is found in the sequence.
- **Not in (`not in`)**: Returns `True` if the value is not found in the sequence.

Example:

```
list1 = [1, 2, 3, 4, 5]

print(3 in list1)     # Output: True
print(6 not in list1) # Output: True
```

viii) Identity Operators

Identity operators are used to compare the memory locations of two objects.

- **Is (`is`)**: Returns `True` if both variables point to the same object.
- **Is not (`is not`)**: Returns `True` if both variables do not point to the same object.

Example:

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a is b)        # Output: False (Different objects in memory)
print(a is not b)   # Output: True

c = a
print(a is c)        # Output: True (Same object in memory)
print(a is not c)   # Output: False
```

4. Write a program to add and multiply complex numbers

Ans:

Program: `complex_operations_no_functions.py`

```
# complex_operations_no_functions.py

if __name__ == "__main__":
    try:
        # Input two complex numbers from the user
        # User inputs are in the format: "real+imagj" or "real-imagj"
        complex1 = complex(input("Enter the first complex number (e.g.,
3+4j): "))
        complex2 = complex(input("Enter the second complex number (e.g., 1-
2j): "))

        # Add two complex numbers
        sum_result = complex1 + complex2

        # Multiply two complex numbers
        product_result = complex1 * complex2

        # Display results
        print(f"Sum of {complex1} and {complex2} is: {sum_result}")
        print(f"Product of {complex1} and {complex2} is: {product_result}")

    except ValueError:
        print("Please enter valid complex numbers.")
```

Run: python complex_operations_no_functions.py

Output 1:

Enter the first complex number (e.g., 3+4j): 2+3j

Enter the second complex number (e.g., 1-2j): 4+5j

Sum of (2+3j) and (4+5j) is: (6+8j)

Product of (2+3j) and (4+5j) is: (-7+22j)

Output 2:

```
Enter the first complex number (e.g., 3+4j): 1+2j
Enter the second complex number (e.g., 1-1j): 2-1j
Sum of (1+2j) and (2-1j) is: (3+1j)
Product of (1+2j) and (2-1j) is: (4+1j)
```

Program: `complex_operations.py`

```
# complex_operations.py

def add_complex(c1, c2):
    """
    Add two complex numbers.
    """
    return c1 + c2

def multiply_complex(c1, c2):
    """
    Multiply two complex numbers.
    """
    return c1 * c2

if __name__ == "__main__":
    # Input two complex numbers from the user
    try:
        # User inputs are in the format: "real+imagj" or "real-imagj"
        complex1 = complex(input("Enter the first complex number (e.g.,
3+4j): "))
        complex2 = complex(input("Enter the second complex number (e.g., 1-
2j): "))

        # Perform operations
        sum_result = add_complex(complex1, complex2)
        product_result = multiply_complex(complex1, complex2)

        # Display results
        print(f"Sum of {complex1} and {complex2} is: {sum_result}")
        print(f"Product of {complex1} and {complex2} is: {product_result}")

    except ValueError:
        print("Please enter valid complex numbers.")
```

Explanation

1. **Function Definitions**:
   - **`add_complex(c1, c2)`**: Adds two complex numbers.
   - **`multiply_complex(c1, c2)`**: Multiplies two complex numbers.

2. **Main Block**:
    - Takes two complex numbers as input from the user.
    - Computes the sum and product using the defined functions.
    - Prints the results.

How to Run the Program

1. **Save the Program**: Save the code above in a file named `complex_operations.py`.
2. **Execute the Program**:
    - Open your command line or terminal.
    - Navigate to the directory where `complex_operations.py` is saved.
    - Run the program using Python by typing:

      ```
      python complex_operations.py
      ```

3. **Input Complex Numbers**: Enter two complex numbers in the format `real+imagj` or `real-imagj` when prompted.

Example Execution

Example 1:

```
Enter the first complex number (e.g., 3+4j): 2+3j
Enter the second complex number (e.g., 1-2j): 1-2j
Sum of (2+3j) and (1-2j) is: (3+1j)
Product of (2+3j) and (1-2j) is: (8-1j)
```

Example 2:

```
Enter the first complex number (e.g., 3+4j): 5-2j
Enter the second complex number (e.g., 1+3j): -3+4j
Sum of (5-2j) and (-3+4j) is: (2+2j)
Product of (5-2j) and (-3+4j) is: (-23+14j)
```

Write a program to print multiplication table of a given number.

Program: `multiplication_table_no_functions.py`

```
# multiplication_table_no_functions.py

# Input number from the user
try:
    number = int(input("Enter a number to print its multiplication table: "))

    # Print the multiplication table
    print(f"Multiplication table for {number}:")
    for i in range(1, 11):
        result = number * i
```

```
        print(f"{number} x {i} = {result}")

except ValueError:
    print("Please enter a valid integer.")
```

Explanation

1. **Input Handling**:
   - **`number = int(input(...))`**: Prompts the user to enter an integer and converts the input to an integer.

2. **Multiplication Table Generation**:
   - **`for i in range(1, 11)`**: Loops from 1 to 10.
   - **`result = number * i`**: Computes the product.
   - **`print(f"{number} x {i} = {result}")`**: Prints each line of the multiplication table.

3. **Error Handling**:
   - **`except ValueError`**: Catches and handles the case where the input is not a valid integer.

How to Run the Program

1. **Save the Program**: Save the code above in a file named `multiplication_table_no_functions.py`.
2. **Execute the Program**:
   - Open your command line or terminal.
   - Navigate to the directory where `multiplication_table_no_functions.py` is saved.
   - Run the program using Python by typing:

     ```
     python multiplication_table_no_functions.py
     ```

3. **Input Number**: Enter an integer when prompted.

Example Execution

Example 1:

```
Enter a number to print its multiplication table: 7
Multiplication table for 7:
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
```

7 x 9 = 63
7 x 10 = 70