

Chapter 1: Introduction to Python

1.1 Introduction to Programming

1.1.1 Meaning of Programming

- * Programming is the process of **writing instructions** that a computer can understand and execute.
- * These instructions are used to **solve problems**, perform tasks, and create useful applications.
- * In simple words, programming is the way of **communicating with a computer** to get a desired result.



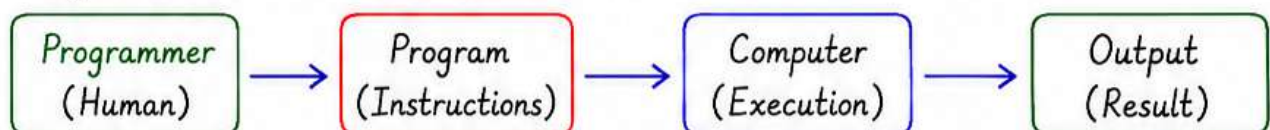
Key Points:

- ✓ Programming gives instructions to the computer.
- ✓ It is used to solve real-life problems.
- ✓ It helps in building software, websites, games, and many more applications.



1.1.2 Programs and Programming Languages

- * A **program** is a set of instructions written in a programming language to perform a specific task.
- * A **programming language** is a formal language used to write programs that can be understood by a computer.
- * There are **many programming languages** available, such as Python, C, C++, Java, etc.



Examples:

- A program to add two numbers.
- A program to calculate the area of a circle.
- A program to display "Hello, World!" on the screen.

Programs are written in a language, converted to machine code, and executed by the computer.

1.1.3 Problem-Solving with Computers

- * Computers follow a **step-by-step approach** to solve problems.
- * A problem is solved by breaking it into **smaller steps**.
- * This process is called an **algorithm**.
- * Once the **algorithm** is ready, it is written in a **programming language** and executed by the computer.
- * The computer processes the instructions and gives the **output** as the solution.



Steps in Problem-Solving:



Key Points:

- ✓ Computers are very fast, accurate and can work for long hours.
- ✓ They do not have intelligence; they follow the instructions given by us.
- ✓ Good programming makes a computer solve complex problems easily.



1.1.4 Real-World Applications of Programming

- * Programming is used in almost every field today. Some common applications are:



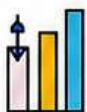
- **Web Development** – Websites like Amazon, Flipkart, etc.



- **Mobile Apps** – Apps like Instagram, WhatsApp, TikTok, etc.



- **Games** – All types of 2D and 3D games are created using programming.



- **Data Science** – Analyzing data, making predictions and reports.



- **Artificial Intelligence** – Machines can think, learn and make decisions.



- **Banking & Finance** – Used in online banking, transactions and security.



- **Healthcare** – Used in patient records, medical devices and research.



- **Automobiles** – Used in self-driving cars and advanced safety systems.

Key Takeaway:



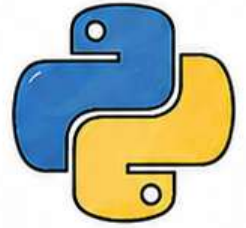
Programming helps us solve problems, build useful applications and make the world **smarter, faster and better**.



1.2 Introduction to Python

1.2.1 What Is Python?

- * Python is a **high-level**, **interpreted**, general-purpose programming language.
- * It was designed to be **easy to read** and **easy to write**.
- * Python supports multiple programming paradigms like **procedural**, **object-oriented** and **functional** programming.
- * It has a large standard library and a vibrant community.
- * Python is used for web development, data science, artificial intelligence, automation, testing and many more.

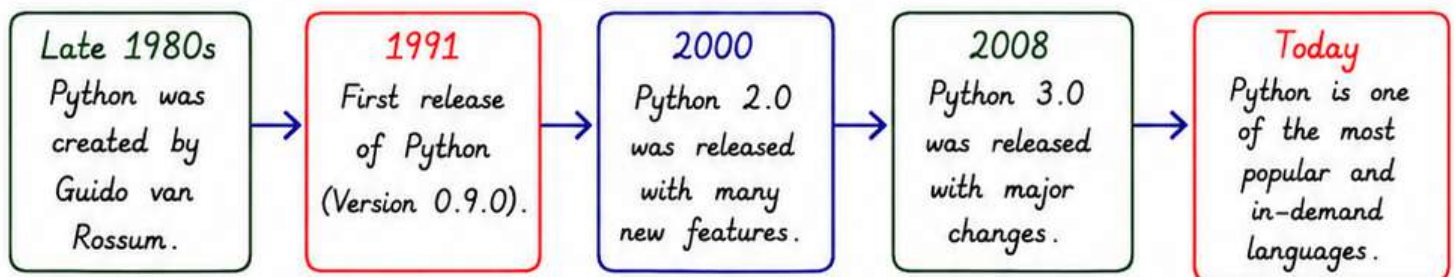


Key Points:

- ✓ Python is simple, powerful and versatile.
- ✓ It is beginner-friendly and widely used.
- ✓ It runs on many platforms like Windows, Linux and macOS.



1.2.2 History and Evolution of Python



- * Python **3.x** is the latest and most widely used version.

1.2.3 Features of Python



- * **Easy to Learn and Use** - Python syntax is simple and readable.



- * **High-Level Language** - Python is closer to human language.



- * **Interpreted Language** - Python code is executed line by line.



- * **Cross-Platform** - Python runs on Windows, Linux, macOS and other platforms.



- * **Large Standard Library** - Python comes with a vast collection of built-in modules.



- * **Versatile** - Used in web development, data science, AI, automation, testing and more.



- * **Open Source** - Python is open source and free to use.



- * **Community Support** - Huge community support and many resources are available.

Key Takeaway:



Python is **simple**, **powerful**, **flexible** and widely used in various domains, making it a great choice for beginners and professionals alike.



1.2.4 Advantages of Python

- *  **Easy to Learn and Use** - Python has simple syntax similar to English, making it easy for beginners.
- *  **High Productivity** - Python allows developers to write and test code quickly, which increases productivity.
- *  **Versatile Language** - Python supports multiple domains like web development, data science, AI, automation, testing, etc.
- *  **Large Standard Library** - Python comes with a huge collection of built-in modules and functions.
- *  **Cross-Platform** - Python programs can run on Windows, Linux, macOS and other platforms without changes.
- *  **Open Source and Community Support** - Python is open source and has a large, helpful community.
- *  **Reliable and Secure** - Python is a reliable language with strong security and active support.
- *  **Great Career Opportunities** - Python is in high demand and offers many career opportunities.

Key Takeaway:



Python is a **powerful, easy to learn, versatile** and **widely used** language, making it a **great choice** for beginners and professionals.

1.2.5 Limitations of Python

- *  **Slower Performance** - Python is slower than compiled languages like C, C++ or Java.
- *  **High Memory Usage** - Python consumes more memory compared to some other languages.
- *  **Not Ideal for Mobile Development** - Python is not the first choice for mobile app development.
- *  **Limited in Some Applications** - Python may not be suitable for real-time systems or low-level programming.
- *  **Global Interpreter Lock (GIL)** - GIL in CPython prevents true parallel execution of threads.

Key Takeaway:



Python has some **limitations** like speed and memory usage, but its **simplicity** and **versatility** make it a **widely preferred** language.



1.3 Applications of Python

1.3.1 Python in Software Development

- * Python is widely used for building **web applications, desktop applications** and **APIs**.
- * Frameworks like **Django, Flask** and **FastAPI** make development fast and easy.
- * It supports multiple programming paradigms (**object-oriented, procedural** and **functional**).
- * Python has excellent libraries and tools that help developers write clean and efficient code.
- * It is used by companies like **Google, YouTube, Instagram, Dropbox** and **Spotify**.

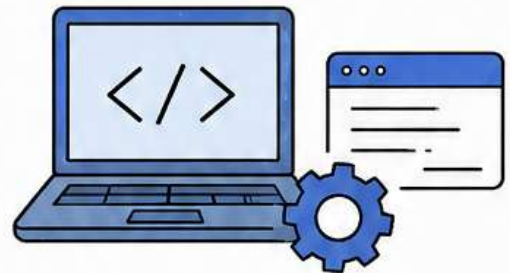
Key Takeaway:



Python simplifies software development by providing **powerful frameworks** and **tools**, making it a top choice for developers.

Examples :

- Web development (using Django, Flask)
- Desktop GUI applications (using Tkinter, PyQt)
- RESTful APIs (using FastAPI)
- Game development (using Pygame)
- Automation tools and scripts
- CMS platforms, E-commerce websites
- Social media and blogging platforms



1.3.2 Python for Data Science

- * Python is one of the most popular languages in **data science**.
- * It has powerful libraries for **data analysis, manipulation** and **visualization**.
- * It helps in extracting insights from data and making **data-driven decisions**.
- * Used in **statistics, data mining, big data** and **business intelligence**.

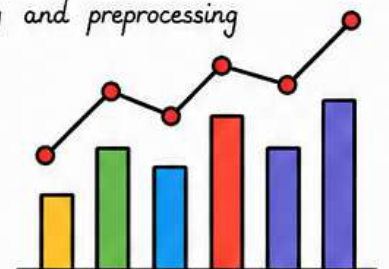
Key Takeaway:



Python provides the best libraries and tools to **collect, analyze, visualize** and **interpret** data effectively.

Examples :

- Data analysis (using Pandas, NumPy)
- Data visualization (using Matplotlib, Seaborn)
- Big data processing (using PySpark)
- Business intelligence and reporting
- Data cleaning and preprocessing



1.3.3 Python for Artificial Intelligence and Machine Learning

- * Python is the most popular language in **AI and ML** development.
- * It has **simple syntax**, excellent libraries and a large community support.
- * Python helps in building intelligent systems that can **learn from data** and make predictions.
- * It is used in tasks like **image recognition**, **natural language processing**, **speech recognition** and more.
- * Python works well with **data analysis**, **statistics** and **visualization** tools.

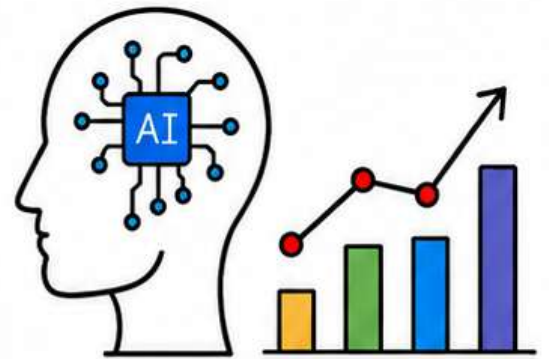
Key Takeaway:



Python provides **powerful libraries** and tools that make it ideal for building **smart systems** and solving complex problems.

Examples:

- Predicting house prices
- Image classification
- Chatbots and virtual assistants
- Speech to text conversion
- Recommendation systems
- Fraud detection in banking
- Self-driving car technology



1.3.4 Python for Web Development

- * Python is widely used for web development due to its **simplicity** and **productivity**.
- * It supports both **front-end** and **back-end** development.
- * Python frameworks make it easy to build **secure**, **dynamic** and **scalable** web applications.
- * It can connect with **databases**, handle user requests and manage **server-side logic**.

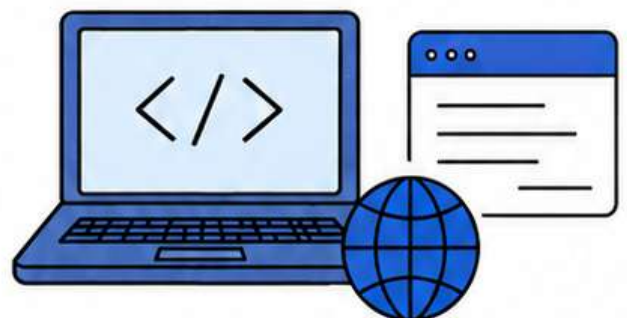
Key Takeaway:



Python frameworks and tools help developers build **fast**, **reliable** and **feature-rich** websites with less effort.

Examples:

- E-commerce websites
- Blogging platforms
- Social networking sites
- Content management systems
- Online booking systems
- RESTful web services



In short, Python is a versatile language used in **AI**, **ML** and **web development**, making it a top choice in today's technology world.

1.3.5 Python for Automation

- * **Automation** means using technology to perform tasks automatically with minimal human intervention.
- * Python is widely used for automation because it is **simple, powerful** and has many libraries.
- * It helps in **saving time, reducing errors** and **increasing productivity**.
- * Python can automate tasks like **file handling, data entry, web scraping, email sending, testing** and **more**.

Example: Automating Email Sending

```
import smtplib
server = smtplib.SMTP('smtp.gmail.com', 587)
server.starttls()
server.login('your_email@gmail.com', 'password')
server.sendmail('from@gmail.com', 'to@gmail.com',
                'Hello! This is an automated email.')
server.quit()
```

Key Takeaway:



Python makes automation **easy** and **efficient**, allowing you to focus on important work while Python handles the repetitive tasks.

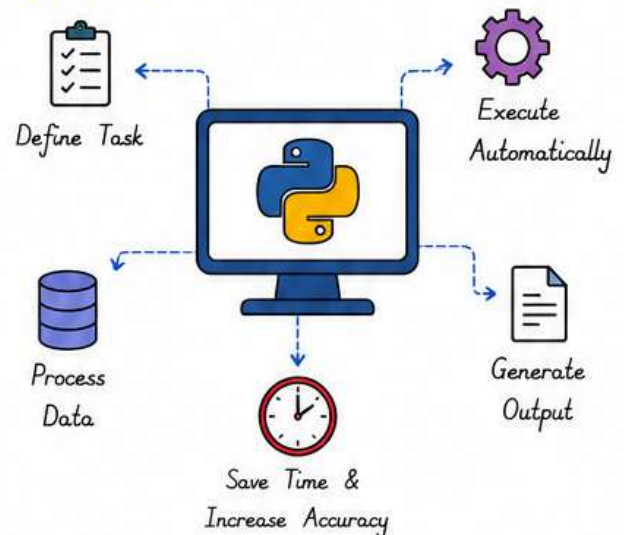
Popular Python Libraries for Automation:

Library	Use in Automation
Selenium	For web automation and testing
BeautifulSoup	For web scraping and data extraction
PyAutoGUI	For GUI automation and control
Schedule	For task scheduling and job automation
Pandas	For data handling and automation
OpenPyXL	For Excel file automation
smtplib	For sending emails automatically
os and shutil	For file and folder management

Applications:

- Automating repetitive office tasks
- Web scraping and data collection
- Sending emails and notifications
- File and folder management
- Data entry in databases or sheets
- Test automation
- Report generation

How Automation Works with Python

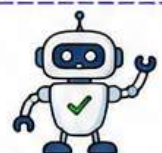


Benefits of Python Automation:

- **Saves time** and effort
- **Reduces** human errors
- **Increases** productivity
- Handles **large volume** of tasks
- Works **24/7** without fatigue
- Improves **accuracy** and **reliability**
- **Cost-effective** solution



Python is a **powerful** tool for automation. It **simplifies** complex and **repetitive tasks**, improves **efficiency** and helps in achieving more with **less effort**.



1.3.6 Python for Robotics and IoT

- * Python plays a key role in **Robotics** and **IoT** (Internet of Things) due to its simplicity and strong library support.
- * It helps in **controlling robots**, **collecting sensor data**, **automating devices** and building smart systems.
- * Python can communicate with sensors, motors, cameras and other hardware components.
- * It is widely used in **smart homes**, **industrial automation**, **drones** and wearable devices.

Example : Reading Temperature from DHT11 Sensor

```
import Adafruit_DHT
sensor = Adafruit_DHT.DHT11
pin = 4
humidity, temperature =
    Adafruit_DHT.read_retry(sensor, pin)
if humidity is not None and temperature is not None:
    print('Temperature :', temperature, '°C')
    print('Humidity :', humidity, '%')
else:
    print('Failed to read from sensor!')
```

Popular Python Libraries for Robotics & IoT :

Library	Use
RPi.GPIO	For Raspberry Pi GPIO control
OpenCV	For computer vision and image processing
Adafruit CircuitPython	For hardware interaction with sensors
Paho-MQTT	For IoT communication using MQTT protocol
PySerial	For serial communication with devices

★ Python is a **powerful** tool for Robotics and IoT. It simplifies hardware interaction, data processing and automation, making it an ideal choice for building **intelligent** and **smart systems**.

Applications :



Robot navigation and control



Sensor data collection and monitoring



Smart home automation



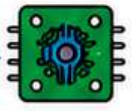
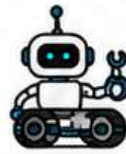
Industrial automation



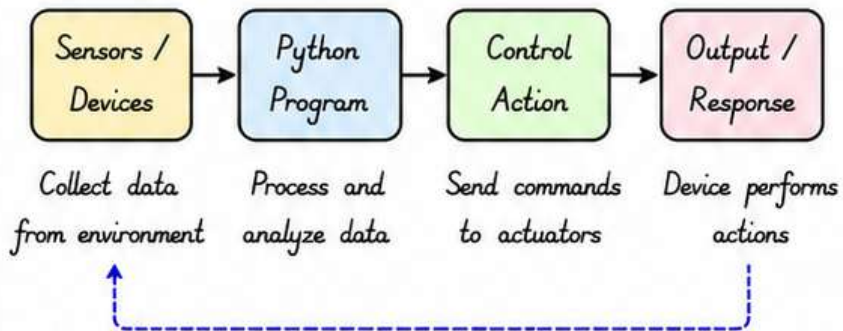
IoT device communication



AI based robotic systems



How Python Works in Robotics and IoT



Key Takeaway :



Python, combined with robotics and IoT technologies, helps in creating **smart**, **connected** and **automated** systems that improve our daily life and solve real-world problems efficiently.



1.3.7 Python for Engineering Applications

- * Python is extensively used in various **engineering** domains due to its simplicity and powerful libraries.
- * It helps engineers in **simulation, analysis, modeling, design optimization** and **data visualization**.
- * Python reduces development time and **improves accuracy** in engineering solutions.
- * It can handle complex **mathematical computations, data analysis** and **real-world engineering problems** efficiently.

Example: Solving a Quadratic Equation

```
import math
a = 1
b = -5
c = 6
discriminant = b**2 - 4*a*c
if discriminant >= 0:
    x1 = (-b + math.sqrt(discriminant)) / (2*a)
    x2 = (-b - math.sqrt(discriminant)) / (2*a)
    print('Roots are:', x1, 'and', x2)
else:
    print('No real roots exist!')
```

Popular Python Libraries for Engineering:

Library	Use in Engineering
NumPy	For numerical computations and array operations
SciPy	For scientific and technical computing
Matplotlib	For plotting graphs and visualizations
Pandas	For data analysis and engineering data handling
SymPy	For symbolic mathematics and equation solving
OpenCV	For image processing and computer vision
PySerial	For communication with hardware devices
CAD Libraries (ezdxf, pyautocad)	For CAD automation and drawing handling

Key Takeaway:



Python is a **powerful** tool for engineers. It simplifies **complex calculations, simulations** and **data analysis**, enabling efficient **design, testing** and **optimization** across multiple engineering fields.



Applications:



Mechanical Engineering - Simulation, CAD automation, stress analysis



Electrical Engineering - Circuit analysis, signal processing, power systems



Civil Engineering - Structural analysis, load calculation, design automation



Chemical Engineering - Process simulation, reaction kinetics

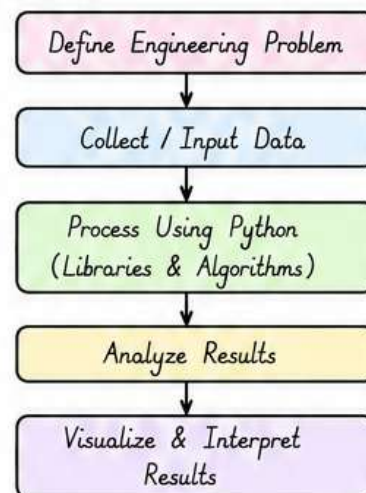


Electronics & Communication - Data analysis, IoT, embedded systems



Data Analysis & Visualization - Plotting results, performance comparison

Engineering Problem Solving with Python



Advantages:

- **Easy** to learn and use
- **Large** collection of engineering libraries
- **Cross-platform** and open source
- **Saves** time and cost
- Improves **accuracy** and **efficiency**



1.4 Python and Other Programming Languages

- * Python is a high-level, interpreted and general-purpose language. It is often compared with other popular languages like C and C++.
- * Each language has its own strengths and is suitable for different types of applications.

Why Compare ?

- Helps us understand the features, advantages and limitations.
- Helps in choosing the right language for the right problem.
- Helps programmers to switch or learn new languages easily.

1.4.1 Python Compared with C

Feature	Python	C
Type of Language	High-level, <i>interpreted</i>	Low-level, <i>compiled</i>
Syntax	<i>Simple</i> and easy to read	<i>Complex</i> and harder to read
Execution Speed	<i>Slower</i> (interpreted)	<i>Faster</i> (compiled)
Memory Management	<i>Automatic</i> (Garbage Collection)	<i>Manual</i> (using malloc, free)
Platform Independence	<i>Yes</i> (Write once, run anywhere)	<i>No</i> (Needs recompilation)
Code Length	<i>Shorter</i>	<i>Longer</i>
Ease of Learning	<i>Easy</i> for beginners	<i>Difficult</i> for beginners
Use Cases	Web, AI, Data Science, Automation, Scripting, etc.	System programming, OS, Embedded systems, etc.

Conclusion :

Python is *easier to learn* and use, while C is better for *system programming* and *performance-critical* applications.



1.4.2 Python Compared with C++

Feature	Python	C++
Type of Language	High-level, <i>interpreted</i>	High-level, <i>compiled</i>
Syntax	Very <i>simple</i> and clean	<i>Complex</i> (many features)
Execution Speed	<i>Slower</i>	<i>Faster</i>
Memory Management	<i>Automatic</i> (Garbage Collection)	<i>Manual</i> and <i>Automatic</i> (new, delete or smart pointers)
OOP Support	<i>Yes</i> (but simpler)	<i>Yes</i> (Advanced and powerful)
Code Length	<i>Shorter</i>	<i>Longer</i>
Ease of Learning	<i>Easy</i>	<i>Moderate to Difficult</i>
Use Cases	Web, AI, Machine Learning, Data Science, Automation, etc.	Game development, System software, High-performance applications, etc.

Conclusion :

Python is better for *productivity* and *rapid* development, while C++ is better for *performance* and *large scale* applications.



Key Takeaway :

Python focuses on *simplicity* and *readability*, while C and C++ focus on *performance* and *system-level programming*. The choice of language depends on the *problem*, *performance needs* and developer *preference*.

1.4.3 Python Compared with Java

- * Python and Java are both popular programming languages, but they have many differences.
- * The table below shows a quick comparison between Python and Java.

Advantages of Python over Java:

- Simpler and easier to learn
- Less code, more productivity
- Interpreted - no compilation needed
- Large number of built-in libraries
- Great for scripting, AI, ML and data science

Feature	Python	Java
Type	Interpreted	Compiled
Syntax	Simple and easy to read	More complex and verbose
Code Length	Shorter code	Longer code
Speed	Slower than Java	Faster than Python
Memory Usage	Higher memory usage	Lower memory usage
Platform	Platform independent	Platform independent (JVM required)
Use Cases	AI, ML, Data Science, Web, Automation, Scripting	Enterprise applications, Android apps, Web, Banking systems
Learning Curve	Easy for beginners	Moderate

Example: Hello World Program

Python:



```
# Hello World in Python
print("Hello, World!")
```

Java:

```
// Hello World in Java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```



Key Takeaway:



Python is easier to learn and write, while Java is faster and better for large-scale applications.

When to Use?



Use Python when you want quick development, automation, AI/ML or data analysis.



Use Java when you need high performance, security and enterprise-level applications.



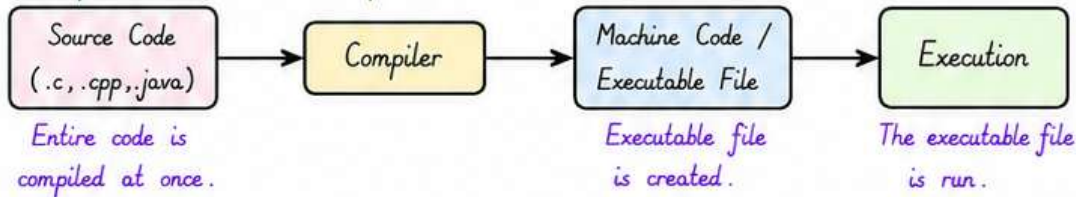
Both Python and Java are powerful languages. The choice depends on your project needs and goals.



1.4.4 Compiler and Interpreter

- * A programming language processes the source code and converts it into a form that the computer can understand and execute.
- * This processing is done using two main tools: **Compiler** and **Interpreter**.

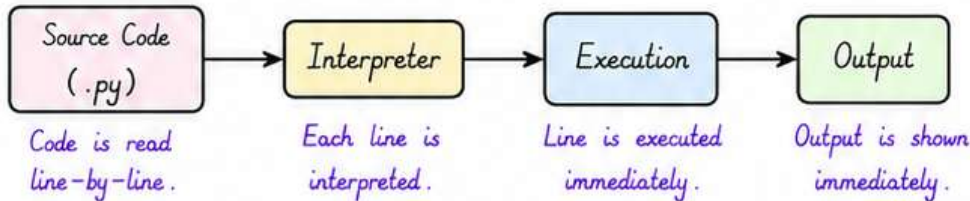
Compilation Process (Compiler)



Compiler :

- Compiles the entire program at once.
- Converts source code into machine code.
- Generates an executable file.
- Errors are shown after compilation.
- Examples : C, C++, Java

Interpretation Process (Interpreter)



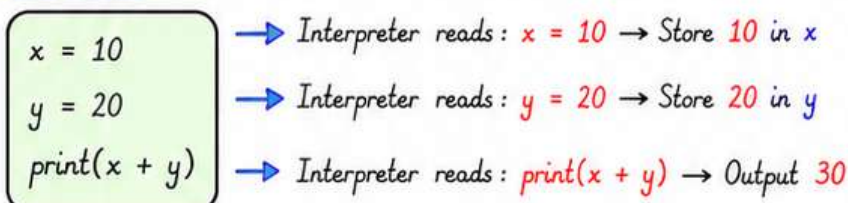
Interpreter :

- Interprets the program line-by-line.
- Does not create an executable file.
- Stops execution if an error occurs.
- Errors are shown immediately.
- Examples : Python, JavaScript, PHP

Comparison Table :

Feature	Compiler	Interpreter
Meaning	Converts entire code into machine code at once.	Converts and executes code line-by-line .
Execution	Creates an executable file .	No executable file is created.
Speed	Faster (after compilation).	Slower (line-by-line execution).
Error Handling	Shows all errors after compilation .	Stops at first error and shows it immediately .
Memory Usage	Less memory during execution.	More memory during execution.
Portability	Executable runs on any machine (platform independent).	Source code needed every time (platform dependent).
Examples	C, C++, Java, Go	Python, JavaScript, PHP, Ruby

Example in Python :



Key Takeaway :



Both compiler and interpreter are important. Compiler gives **speed** and **efficiency**, while interpreter gives **flexibility** and **ease of use**.

1.4.5 Interpreted Nature of Python

- * Python is an interpreted language, not a **compiled** one.
- * It executes code line-by-line using the **Python Interpreter**.
- * This makes Python **easy to write, test and debug**.
- * You **don't** need to **compile** the program before running it.
- * Python shows **errors immediately** at runtime, making debugging simpler.
- * This also makes Python **platform independent**.
- * The interpreted nature provides **flexibility** and **portability**.

Key Takeaway:



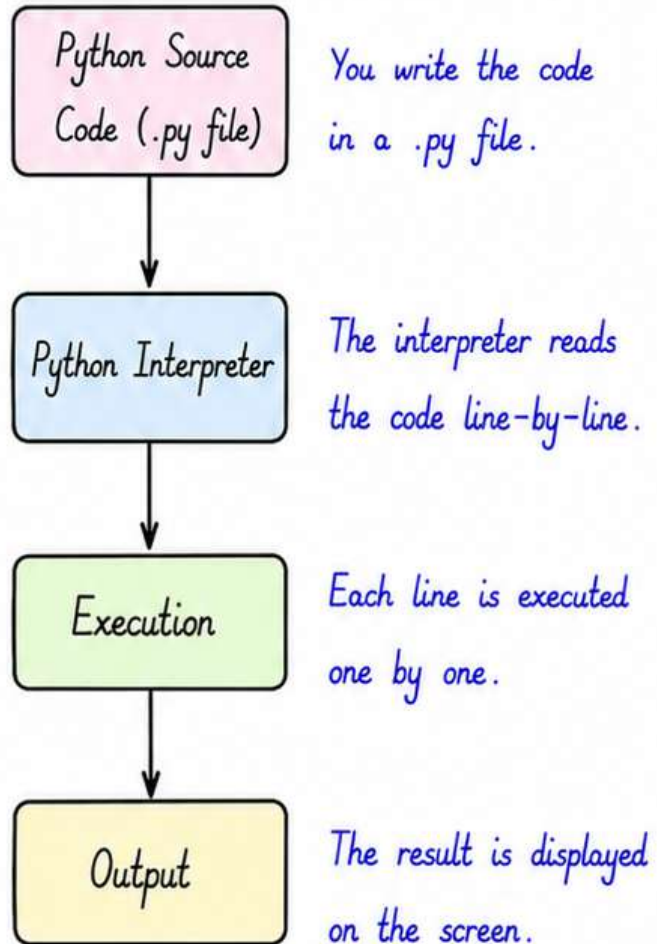
Python's interpreted nature makes it **simple, flexible** and **beginner-friendly**, while also being powerful and efficient.



In short, Python's interpreted nature allows **quick development**, **easy debugging** and **cross-platform execution**, making it one of the most preferred programming languages.

Examples:

- How Python code is executed:



- Example Code:

```
print("Hello, Python!")  
x = 10  
print(x + 5)
```

Output:

```
Hello, Python!  
15
```

1.5 Python Program Execution

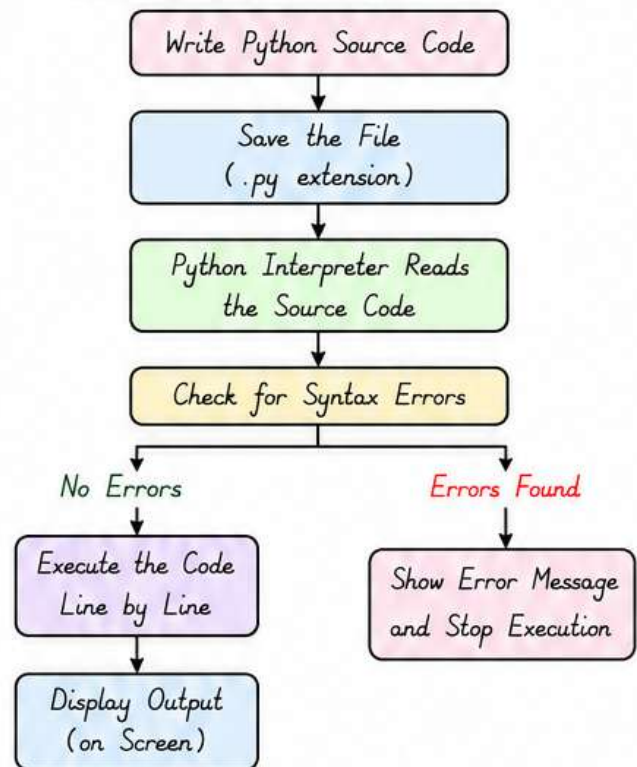
1.5.1 How Python Executes a Program

- * Python executes a program step-by-step from top to bottom.
- * First, the **source code** is read and checked for **syntax errors**.
- * If there are no errors, Python **interprets** each line and executes it.
- * The **output** (if any) is displayed on the screen.
- * If an **error** occurs, execution stops and an error message is shown.

Key Points:

- Python is an **interpreted** language.
- It does **not compile** the entire program at once.
- It executes code **line-by-line**.
- Errors are reported **immediately**.

Steps in Python Program Execution



1.5.2 Source Code

- * Source code is the **human-readable code** written by the programmer.
- * It is written in a **text file** using a code editor or IDE.
- * In Python, source code files have the **.py** extension.
- * The interpreter **reads** this code and **executes** it.

Summary:

How Python Executes a Program

- Read source code
- Check for errors
- Execute line-by-line
- Display output or error message



Example of Source Code:

```
1 # This program adds two numbers → Comment
2 a = 10
3 b = 20 } → Variables
4 sum = a + b → Computation
5 print("The sum is:", sum) → Output Statement
```

Note:

The source code is saved with **.py** extension and run using the Python interpreter.

Source Code

- Written by programmer
- Saved as **.py** file
- Contains instructions for the program
- Read and executed by Python interpreter

1.5.3 Bytecode

- * Bytecode is an **intermediate code** generated by the Python interpreter.
- * It is not machine code, but a **low-level** instruction set for the Python Virtual Machine (PVM).
- * Python **source code** is first converted into **bytecode**, and then executed by the PVM.
- * Bytecode is **platform-independent**, so Python programs can run on any system with Python installed.

Key Points:

- Bytecode is generated **automatically**.
- It is stored in memory as **.pyc** file.
- It is executed by the **Python Virtual Machine**.
- It makes Python **portable** and **efficient**.

Example:

Python Source Code (hello.py)

```
a = 10
b = 20
sum = a + b
print(sum)
```



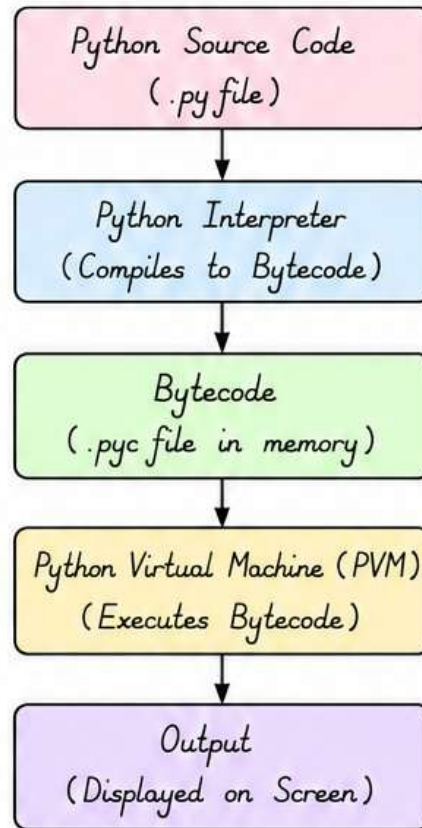
Python Interpreter
Converts to Bytecode

Bytecode (example output)

```
0 LOAD_CONST      1 (10)
2 STORE_NAME      0 (a)
4 LOAD_CONST      2 (20)
6 STORE_NAME      1 (b)
8 LOAD_NAME       0 (a)
10 LOAD_NAME      1 (b)
12 BINARY_ADD
14 STORE_NAME      2 (sum)
16 PRINT_EXPR
18 LOAD_CONST      0 (None)
20 RETURN_VALUE
```

These are
low-level
instructions
understood
by PVM.

How Bytecode Works in Python



Summary:



Bytecode is the **intermediate code** between source code and execution. The Python interpreter converts source code into bytecode, and the Python Virtual Machine executes it to produce output.

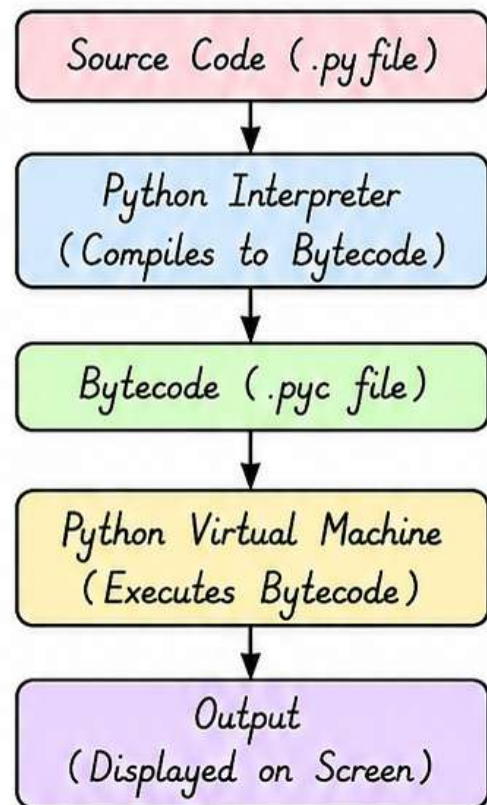
1.5.4 Python Virtual Machine

- * Python Virtual Machine (PVM) is the engine that **executes** Python bytecode.
- * It is a part of the **Python interpreter** and runs on any system where Python is installed.
- * PVM takes **bytecode** as input and executes it line by line to produce **output**.
- * It handles **memory management**, data operations, control flow and built-in functions.

Key Points :

- PVM is **platform-independent**.
- It executes **bytecode**, not source code.
- Same bytecode can run on **different operating systems**.
- It ensures **safety**, **memory management** and **efficient execution**.

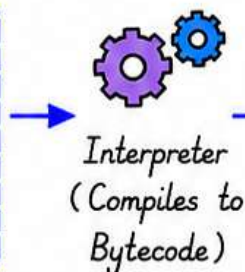
How PVM Works



Example :

Python Source Code

```
a = 5
b = 10
c = a + b
print(c)
```



Bytecode (.pyc)

```
0  LOAD_CONST    1 (5)
2  STORE_NAME    0 (a)
4  LOAD_CONST    2 (10)
6  STORE_NAME    1 (b)
8  LOAD_NAME     0 (a)
10 LOAD_NAME     1 (b)
12 BINARY_ADD
14 STORE_NAME    2 (c)
16 LOAD_NAME     2 (c)
18 PRINT_EXPR
20 LOAD_CONST    0 (None)
22 RETURN_VALUE
```

PVM Executes

15
Output

Summary :



Python Virtual Machine is the **core engine** of Python. It executes **bytecode** produced by the interpreter and makes Python programs **platform-independent** and **efficient**.

1.5.5 Program Execution Flow

- * The program execution flow is the **sequence of steps** followed from writing the code to getting the output.
- * Python reads the program, checks it, executes each instruction, and shows the result.
- * If any error is found, the execution stops and an **error** message is shown.

Key Points:

- Execution starts from the **first line** and goes to the last line.
- Python executes code **line by line**.
- If an **error** occurs, execution stops.
- **Output** is shown only if execution is successful.

Example:

Python Source Code

```
a = 10
b = 20
sum = a + b
print(sum)
```

Execution Steps

- ① Read source code
- ② Check for errors
- ③ Execute `a = 10`
- ④ Execute `b = 20`
- ⑤ Execute `sum = a + b`
- ⑥ Execute `print(sum)`
- ⑦ Display output → 30

Output



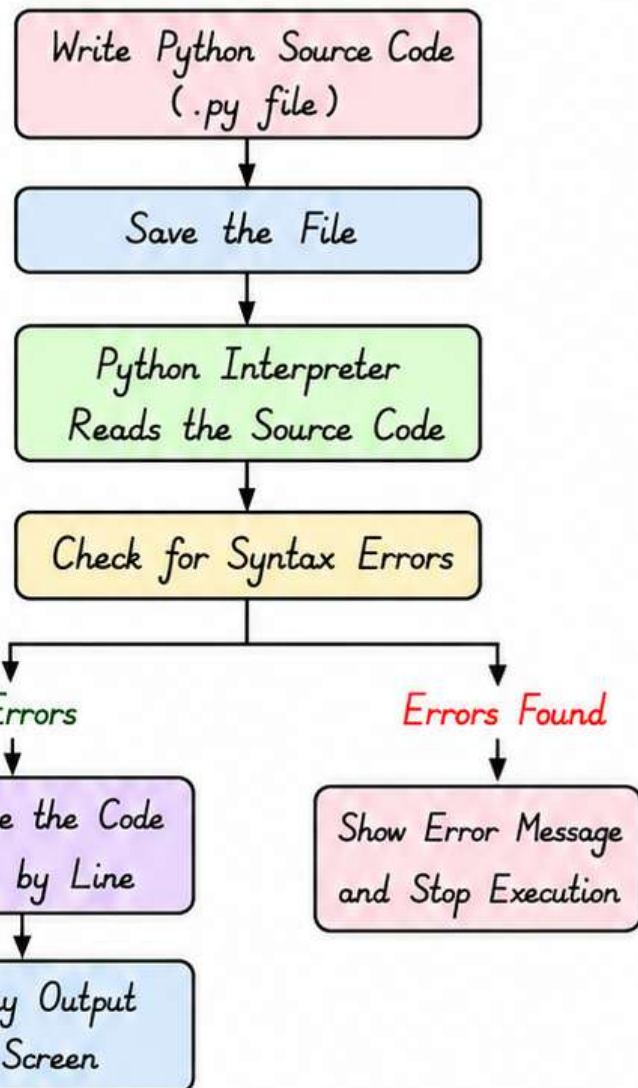
Summary:



Program execution flow is the path followed by a Python program from **source code** to **output**.

Python executes instructions **one by one** and produces the result.

Program Execution Flow



1.6 Python Versions

1.6.1 Major Python Versions

- * Python has evolved through several major versions.
- * The most commonly used major versions are **Python 2** and **Python 3**.

Version	Release Year	Status	Key Notes
Python 1.x	1994	Obsolete	First official release by Guido van Rossum.
Python 2.x	2000	End of Life	Popular for many years. Support ended on 1st January 2020 .
Python 3.x	2008	Active	Current and future version. Regularly updated.

Key Points:

- Python 2 and Python 3 are **not** fully compatible.
- **Python 3** is recommended for all new projects.
- Many libraries and tools now support **only Python 3**.

1.6.2 Python 2

- * Python 2 was released in the year **2000**.
- * It was widely used for a long time.
- * Official support ended on **1st January 2020**.
- * It is **no longer maintained** or updated.

Example (Python 2 Code)

```
a = 10
b = 20
print "Sum is:", a + b    # Print statement
c = 5 / 2                 # Integer division
print "5 / 2 =", c        # Result → 2
```

Note:

- Python 2 code may **not** run in **Python 3** without changes.
- It is recommended to **migrate** old Python 2 code to **Python 3**.

Features of Python 2

- Print statement used without parentheses.
Example : `print "Hello"`
- Division of integers gives **integer** result.
Example : `5 / 2 → 2`
- Unicode string with prefix 'u'.
Example : `u"Hello"`
- Raw input function used.
`raw_input()`
- Has many built-in modules and functions.

Fun Fact !



Python 2 was the backbone of many applications, websites and tools for over a decade.

1.6.3 Python 3

- * Python 3 was released in the year 2008.
- * It is the **current** and **future** version of Python.
- * It is **not backward compatible** with Python 2.
- * Regularly updated with new features and improvements.
- * Recommended for **all new projects** and development.

Features of Python 3

- Simpler and clean syntax.
- `print()` is a function.
- Division of integers gives **float** result.
- Unicode (`str`) is the default.
- Better support for large projects.
- Many new modules and libraries.

Example (Python 3 Code)

```
a = 10
b = 20
print("Sum is:", a + b)  # Print statement
c = 5 / 2
print("5 / 2 =", c)      # Result → 2.5
s = "Hello"
print(s)                 # String
```

Note:

- Python 3 is **actively supported** and receives regular updates.
- Most modern libraries, tools and platforms support **only Python 3**.

1.6.4 Python 2 versus Python 3

Key Differences

Feature	Python 2	Python 3
Release Year	2000	2008
Print Statement	<code>print "Hello"</code>	<code>print("Hello")</code>
Division	$5 / 2 \rightarrow 2$ (integer result)	$5 / 2 \rightarrow 2.5$ (float result)
String Type	ASCII by default	Unicode by default
Input Function	<code>raw_input()</code>	<code>input()</code>
Compatibility	Not compatible with Python 3	Not compatible with Python 2
Support Status	Ended on 1st January 2020	Actively supported
Recommended For	Legacy projects only	All new projects

Summary:



- Python 2 is the older version and is **no longer supported**.
- Python 3 is the **modern version** and should be used for all new development.

1.7 Structure of a Python Program

1.7.1 Python Statements

- * A Python program is made up of **statements**.
- * A statement is a **single instruction** that Python can execute.
- * Each statement performs a **specific task**.
- * Statements can be **simple** or **compound**.
- * Python uses statements to **control the flow of execution** and to perform operations.
- * **Examples**: assignment, print, input, if, for, while functions, etc.

Types of Statements

Type	Description
Simple Statements	Single line instructions that perform a single task. Example : <code>a = 10</code>
Compound Statements	Made up of one or more lines (use indentation). Example : <code>if</code> , <code>for</code> , <code>while</code>

Examples of Statements

- Assignment Statement : `x = 5` # assigns 5 to x
- Print Statement : `print("Hello, Python!")` # prints text
- Input Statement : `name = input("Enter your name:")` # takes input from user
- Expression Statement : `y = x + 10` # expression is evaluated

Key Points

- Every line that Python executes is a **statement**.
- Statements are the **building blocks** of a program.
- Statements can be **simple** (single line) or **compound** (multiple lines).
- **Indentation** is used in compound statements to group instructions.

Example Program (using different statements)

```
a = 10           # Assignment Statement
b = 20           # Assignment Statement
c = a + b        # Expression Statement
print("Sum is:", c) # Print Statement
name = input("Enter your name:") # Input Statement
print("Hello", name) # Print Statement
```

Note :



Statements are the basic units of a Python program. They tell Python what to do step by step.

1.7.2 Code Blocks

- * A code block is a **group of statements** written together and executed as a unit.
- * **Indentation** (spaces or tabs) is used to define a block.
- * All statements in the **same indentation level** belong to the **same block**.
- * A block starts after a **colon (:)** and continues with **indented statements**.
- * Python **does not use { }** to define blocks.

Rules of Code Blocks

- Indentation must be **consistent** within a block.
- A block starts after a **colon (:)**.
- There must be at least one indented statement inside the block.
- When the indentation level decreases, the block ends.
- Python is **indentation sensitive**.

Visual Representation

```
if condition:
    statement 1
    statement 2
else:
    statement 3
    statement 4
```

All statements with the **same indentation** belong to the **same block**.

When indentation decreases, the block terminates.

Example

```
if x > 0:                # If block
    print("Positive")
    print("x is greater than 0")
else:                    # Else block
    print("Non-positive")
```

If Block

Else Block

Another Example (for loop)

```
for i in range(3):       # For block
    print(i)
    print("Hello")
```

For Block

Invalid Example (Indentation Error)

```
if x > 0:
    print("Positive")
    print("x is greater than 0")
else:
    print("Non-positive")
```

✗ Wrong

(Inconsistent indentation)

Remember!

Indentation is the **key** to defining blocks in Python.



Key Takeaway:



- Code blocks help Python **organize** and **control the flow** of execution.
- Proper indentation makes your code **correct**, **readable** and **error-free**.

1.7.3 Indentation

- * Indentation means **spaces or tabs** at the beginning of a line.
- * It is used to **define blocks** of code.
- * All statements in the same indentation belong to the **same block**.
- * Python uses indentation to control the **flow of execution**.
- * Python **does not use** { } (braces) to define blocks.

Key Takeaway:



Proper indentation is must in Python. Wrong indentation leads to **Indentation Error**.

Example

```
if x > 0:           # If block
    print("Positive")
    print("x is greater than 0")
else:               # Else block
    print("Non-positive")
```

} If Block

} Else Block

1.7.4 Case Sensitivity

- * Python is **case sensitive**.
- * It treats uppercase and lowercase letters as **different**.
- * Variable names, function names and keywords must be written in the **correct case**.
- * Even small change in case can change the meaning or cause an **error**.

Example (Observe the difference)

```
age = 20           # valid
Age = 30           # valid
AGE = 40           # valid
print(age)         # prints 20
print(Age)         # prints 30
print(AGE)         # prints 40
```

Examples

- **Variables**
age, Age and AGE are three **different** variables.
- **Functions**
print() and Print() are **not** the same.
- **Keywords**
True, true and TRUE are treated **differently**.

Key Takeaway:



Always use the **correct case** while writing variable names, function names and keywords. Python treats each case as a **different identifier**.

1.7.5 Basic Program Structure

- * Every Python program follows a **basic structure**.
- * It starts with **writing statements**.
- * Statements are grouped into blocks using **indentation**.
- * Python program is executed from **top to bottom**.
- * **Comments** are optional but useful to explain the program.

General Structure

```
# 1. Comments (optional)
# This is a comment

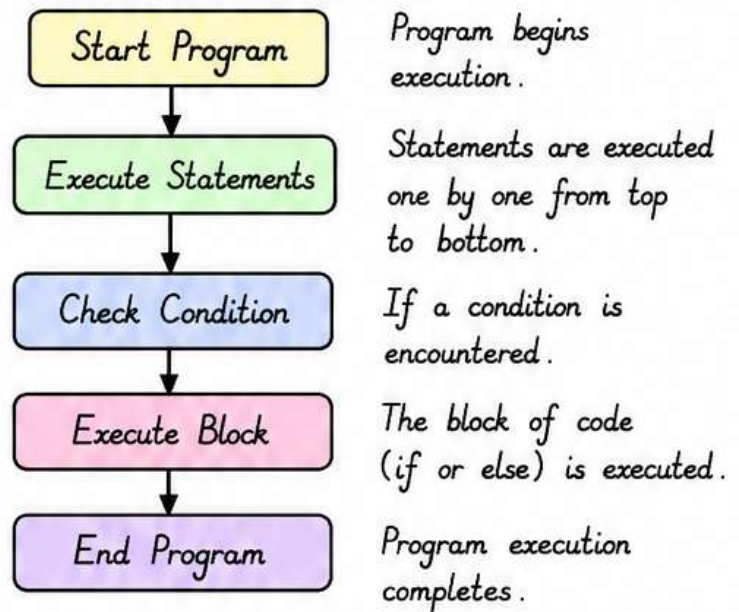
# 2. Statements
statement 1
statement 2

# 3. Indentation for blocks
if condition:
    statement inside block
else:
    another statement
```

Example Program

```
# This program adds two numbers
a = 10          # first number
b = 20          # second number
sum = a + b     # calculate sum
print("Sum is:", sum) # display result
if sum > 20:
    print("Sum is greater than 20")
else:
    print("Sum is 20 or less")
```

Flow of Execution



Key Points

- Python code is executed **line by line**.
- **Indentation** is used to define blocks.
- **Comments** improve code readability.
- Use **proper structure** for error-free programs.
- Always **test** your program with different values.

Example Output

Input:

```
a = 10
b = 20
```

Output:

```
Sum is: 30
Sum is greater than 20
```

Key Takeaway:



A well-structured Python program is **easy to write, understand, debug and maintain**.

1.8 Python Keywords and Identifiers

1.8.1 Python Keywords

- * Keywords are **reserved words** in Python.
- * They have special meaning and cannot be used as **identifiers**.
- * Keywords are part of Python **syntax** and are used to define the structure of a program.
- * All keywords are written in **lowercase**.

List of Python Keywords

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

Example

```
if x > 0:           # 'if' and 'else' are keywords
    print("Positive") # cannot be used as variable names
else:
    print("Non-positive")
```

Remember!

Keywords **cannot** be used as variable names, function names or any other **identifiers**.



Example: Using keyword as identifier

```
if    = 10      ✗
for   = 5       ✗
def   = "hello" ✗
class = "Test"  ✗
```

Why keywords are important?

- They help Python understand the **structure** of the program.
- They are used to perform **specific operations**.
- Using keywords makes the code **meaningful** and **readable**.

Key Takeaway:



- Keywords are **predefined words** with special meaning in Python.
- They **cannot** be used as identifiers (variable names, function names, class names, etc.).

1.8.2 Python Identifiers

- * Identifiers are names given to **variables**, **functions**, **classes** and other objects in Python.
- * They are used to identify different entities in a program.
- * An identifier must follow certain **rules** defined by Python.
- * Identifiers are **case sensitive**.

Rules for Python Identifiers

- An identifier can contain **letters** (A-Z, a-z), **digits** (0-9) and **underscore** (_).
- It must **start** with a **letter** (A-Z, a-z) or an **underscore** (_).
- It cannot start with a **digit**.
- It cannot be a Python **keyword**.
- No **special characters** are allowed (@, #, \$, %, etc.).
- It can be of any **length**.

Valid Identifiers (Examples)

✓ name	# simple identifier
✓ _name	# starts with underscore
✓ name2	# letters followed by digits
✓ total_marks	# underscore in between
✓ calculateSum	# camelCase
✓ _temp_value2	# combination of all rules

Invalid Identifiers (Examples)

✗ 2name	# starts with digit
✗ name!	# special character
✗ total-marks	# hyphen not allowed
✗ for	# Python keyword
✗ class	# Python keyword
✗ name space	# space not allowed

Example Program

```
# Valid identifiers
count = 10           # variable name
_value = 25.5        # starts with underscore
totalAmount = count + _value # camelCase

# Invalid identifiers (causes error)
2count = 5           # starts with digit
total-amount = 100   # hyphen not allowed
for = 20              # keyword cannot be used
```

Why Identifiers are Important?

- They help in writing **meaningful** and **readable** code.
- Good identifiers make the program **easy to understand** and **maintain**.
- They represent the **data** and **actions** in a program.

Remember!

Follow **naming rules** for identifiers to avoid errors and make your code **professional**.



Key Takeaway:



- Identifiers are **user-defined names** for variables, functions, classes, etc.
- They must follow **specific rules**.
- Identifiers are **case sensitive**.
- Good identifiers improve code **readability** and **maintainability**.

1.8.3 Rules for Naming Identifiers.

- * Identifiers are names given to **variables**, **functions**, **classes** and other objects in Python.
- * Python follows certain **rules** for creating valid identifiers.

Rules:

- ① An identifier can contain **letters** (A-Z, a-z), **digits** (0-9) and **underscore** (_).
- ② The **first character** must be a **letter** (A-Z, a-z) or **underscore** (_).
- ③ It cannot start with a **digit**.
- ④ It cannot be a Python **keyword**.
- ⑤ No **special characters** are allowed (@, #, \$, %, etc.).
- ⑥ It can be of any **length**.
- ⑦ Identifiers are **case sensitive**.

Examples (Valid Identifiers)

→ total_marks
→ _value
→ studentName
→ calculateSum
→ age2
→ my_variable_1

Examples (Invalid Identifiers)

- ✗ 2name → starts with digit
- ✗ name! → special character
- ✗ total-marks → hyphen not allowed
- ✗ for → Python keyword
- ✗ name space → space not allowed

Key Takeaway:



Follow these rules to create valid identifiers and write **clean**, **readable** and **error-free** Python programs.

1.8.4 Valid and Invalid Identifiers

* Based on the rules, identifiers can be **valid** or **invalid**.

Valid Identifiers (Examples)

Identifier	Reason
name	Starts with letter
_name	Starts with underscore
name2	Letters followed by digits
total_marks	Underscore in between
calculateSum	CamelCase (no spaces)
_value123	Starts with underscore
Student_Age	Underscore in between
a1b2c3	Combination of letters and digits

Invalid Identifiers (Examples)

Identifier	Reason
2name	Starts with digit
name!	Contains special character
total-marks	Hyphen (-) not allowed
for	Python keyword
class	Python keyword
name space	Spaces not allowed
@value	Special character not allowed
123abc	Starts with digit

Example in Program

Valid Identifiers

```
marks = 95
_count = 10
totalMarks = marks + _count
```

Invalid Identifiers (will cause error)

```
2marks = 50           # starts with digit
total-marks = 40      # hyphen not allowed
for = 10              # keyword cannot be used
name space = "Alex"   # space not allowed
```

Key Takeaway:



- Use **letters**, **digits** and **underscore** (_) to create identifiers.
- Start with a **letter** or **underscore**.
- Do not use Python **keywords** or **special characters**.
- Identifiers are **case sensitive**.
- Follow rules to **avoid errors**.

1.8.5 Python Coding Conventions

- * Coding conventions are a set of guidelines for writing **clean**, **readable** and **consistent** Python code.
- * Following conventions makes code **easier to read**, **understand**, **maintain** and **collaborate**.

General Guidelines

- ① Follow **PEP 8** - the official style guide for Python.
- ② Use **4 spaces** for indentation. (Do not use tabs)
- ③ Limit all lines to a maximum of **79 characters**.
- ④ Use **meaningful** and **descriptive** names.
- ⑤ Keep code **simple** and **easy to understand**.
- ⑥ Add **comments** to explain important logic.
- ⑦ Leave **blank lines** to improve readability.

Examples

- | | | |
|---|--------------------|----------------------|
| ✓ | good_variable = 10 | # good variable name |
| ✓ | calculate_sum() | # good function name |
| ✓ | MyClass | # good class name |
| ✓ | MAX_VALUE = 100 | # good constant name |
| ✗ | GoodVariable = 10 | # not recommended |
| ✗ | calculateSum() | # not recommended |
| ✗ | myclass | # not recommended |
| ✗ | maxValue = 100 | # not recommended |

Comments in Python

- * Use comments to explain **what** the code does, **not how**.
- * Single line comment: `# this is a comment`
- * Multi-line comment: `""" This is
a multi-line comment """`

* **Remember:** Write code for **humans to read**, not just for machines to execute.

Naming Conventions

Identifier Type	Convention
Variable	lower_case_with_underscores
Function	lower_case_with_underscores
Class	CapitalizedWords (PascalCase)
Constant	UPPER_CASE_WITH_UNDERSCORES
Private	<code>_leading_underscore</code>
Module	lower_case_with_underscores
Package	lower_case_with_underscores

Code Layout

- * Use one statement per line.
- * Use blank line between functions and classes.
- * Use spaces around operators:
`a = b + c` (not `a=b+c`)
- * Avoid trailing whitespace.
- * End files with a single newline.
- * Use consistent indentation throughout the code.

Key Takeaway



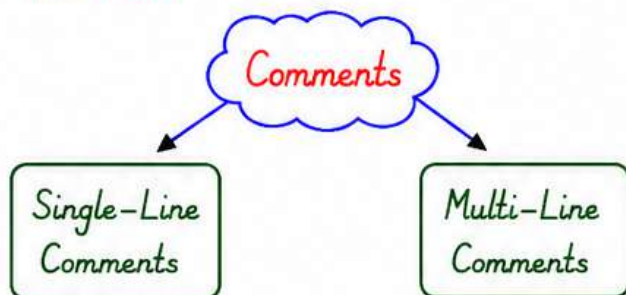
Good coding conventions lead to **clean**, **readable**, **maintainable** and professional Python programs.

1.9 Comments and Documentation

- * Comments are used to **explain** the code. They are ignored by the Python interpreter and are **not executed**.

1.9.1 Comments in Python

- Comments help to make the code **easy to read and understand**.
- They are useful for **describing** what the code does.
- Comments make the code more **maintainable** and help in **debugging**.
- Python supports **single-line** comments and **multi-line** comments.



Example

```
# This is a single-line comment
x = 10          # x is assigned the value 10
y = 20          # y is assigned the value 20
total = x + y    # This line adds x and y
print(total)     # This prints the result
```

Key Points

- Use comments to **explain** the purpose of code.
- Comments improve **readability**.
- They are **ignored** by the interpreter.
- Use **meaningful** comments.
- Keep comments **up-to-date** with the code.

Best Practice



- Write **clear** and **meaningful** comments.
- Do not **over-comment** the code.
- **Update** comments whenever code changes.

Good comments
lead to good code!



1.9.2 Single-Line Comments

- In Python, single-line comments start with a hash symbol (**#**).
- The comment continues till the **end of the line**.
- Used to write **short** explanations.
- Helps to understand **specific line** of code.

Syntax

```
# comment
```

Output

```
30
```

Example (Without Comments)

```
x = 10
y = 20
total = x + y
print(total)
```



Hard to
understand
without
comments.

1.9.3 Multiline Comments

- * Multiline comments are used to write comments that span **multiple lines**.
- * They are **ignored** by the Python interpreter.
- * Useful for **long explanations, notes** or describing blocks of code.

Syntax

```
"""  
This is a  
multiline comment.  
It can span multiple  
lines.  
"""
```

Key Points

- Written using **triple single quotes** (``').
- Can be written using **triple double quotes** (``").
- Can extend over many lines.
- Used to document code and explain logic.
- Helps in better readability and understanding.

Example

```
x = 10  
y = 20  
"""  
This program adds two numbers  
and displays the result.  
It uses multiline comment  
for explanation.  
"""  
total = x + y  
print(total)
```

variables
add numbers
display result

Example Using Triple Double Quotes

```
a = 5  
b = 7  
"""  
This program subtracts b from a.  
It shows the use of triple  
double quotes for comments.  
"""  
result = a - b  
print(result)
```

subtract
print result

When to Use?

- When you need to write **long explanations**.
- When describing a **function, class** or **module**.
- When you want to **temporarily disable** a block of code.
- To add notes or **TODOs** in your program.

Example - Commenting a Block of Code

```
x = 10  
y = 20  
"""  
Below code is for testing purpose.  
It is currently not required.  
So, we are commenting it out.  
"""  
z = x * y  
print(z)  
print("Multiplication Done")
```

Output

Multiplication Done

Summary



- Multiline comments are written using **triple quotes**.
- They are **ignored** by Python.
- Helps in **documenting code** and improving **readability**.
- Can be written using **single quotes** or **double quotes**.

Good comments
make good code!



1.9.4 Docstrings

- * **Docstrings** (Documentation Strings) are used to describe what a **module, function, class** or **method** does.
- * They help developers and users **understand the code easily**.
- * Python uses docstrings for documentation and they can be accessed using **help()**, **__doc__** or IDE tools.

1. What is a Docstring?

A docstring is a string literal written as the first statement inside a module, function, class or method.

It is enclosed in triple quotes:

`"""` or `'''`

2. Where to Use Docstrings?

Location	Purpose
Module	Describes the purpose of the module
Function	Explains what the function does, its parameters and return value
Class	Describes the class and its behaviour
Method	Explains what the method does inside a class

3. Example

(a) Module Docstring

```
"""
This module provides basic math operations.
"""
```

(b) Function Docstring

```
def add(a, b):
    """
    Returns the sum of two numbers.
    Parameters:
        a (int): First number
        b (int): Second number
    Returns:
        int: Sum of a and b
    """
    return a + b
```

(c) Class Docstring

```
class Calculator:
    """
    A simple calculator class.
    It performs basic arithmetic operations.
    """
    def multiply(self, x, y):
        """
        Returns the product of x and y.
        Parameters:
            x (int): First number
            y (int): Second number
        Returns:
            int: Product of x and y
        """
        return x * y
```

4. Accessing Docstrings

- Use **help()** function
- Use **__doc__** attribute
- Use IDE / Editor tooltips
(Hover over the function or class)

5. Benefits of Docstrings

- ✓ Improves code readability.
- ✓ Helps in understanding the purpose of code.
- ✓ Useful for documentation generation (e.g., Sphinx).
- ✓ Helpful for other developers and future reference.

6. Best Practices

- * Write meaningful and clear docstrings.
- * Use triple quotes (`"""` or `'''`).
- * Write docstring as the first statement.
- * Use proper structure: short description, parameters, return value, etc.
- * Keep it updated with the code changes.



Key Takeaway: Docstrings make your code self-explanatory, easy to maintain and professional.

1.9.5 Importance of Program Documentation

- * Program documentation means writing **clear notes** and **explanations** about a program.
- * It helps in **understanding**, **using**, **maintaining** and **improving** a program easily.

Why Documentation is Important ?

- ① Improves **Readability** : Makes code easy to read and understand.
- ② Helps in **Maintenance** : Easy to find and fix errors in the code.
- ③ **Saves Time** : Reduces the time needed to understand or update the program.
- ④ Supports **Teamwork** : Helps other programmers to understand and work on the code.
- ⑤ Better **Learning** : Helps beginners learn from well-documented code.

What to Document ?

Part of Program	What to Document
Program Purpose	What the program does.
Variables	What each variable stores.
Functions	What each function does.
Logic / Steps	Step-by-step explanation of the logic.
Inputs & Outputs	What input is taken and what output is shown.
Errors	Possible errors and how to handle them.

Examples of Good Documentation

```
# This program calculates simple interest
# Input: principal, rate, time
# Output: simple interest
P = float(input("Enter Principal: ")) # Read principal
R = float(input("Enter Rate: "))      # Read rate
T = float(input("Enter Time: "))      # Read time
SI = (P * R * T) / 100                # Calculate SI
print("Simple Interest = ", si)       # Display result
```

Bad Documentation (Avoid)

- * No comments in the code.
- * Wrong or confusing comments.
- * Outdated documentation.
- * Too much or unnecessary information.
- * Inconsistent style of writing.

Types of Comments in Python

- * Single line comment: `# This is a comment`
Used to explain a single line.
- * Multi-line comment :
`""" This is
a multi-line comment """`
Used to explain multiple lines.

Best Practices

- ✓ Write comments for important parts only.
- ✓ Keep comments clear and simple.
- ✓ Update comments whenever code changes.
- ✓ Use meaningful names for variables and functions.

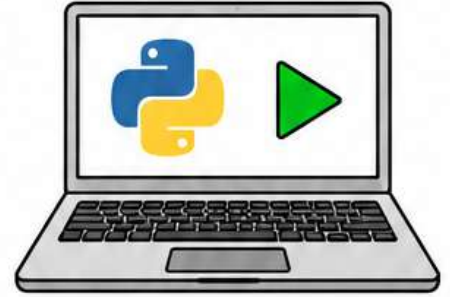


Key Takeaway

Good documentation makes your code **clean**, **easy to understand**, **easy to manage** and **reliable**.

1.10 First Python Program

- * A computer program is a **set of instructions** written to perform a specific task.
- * Let us write and run our first Python program.



1.10.1 Writing the First Program

- Every Python program starts with writing the code, saving it with **.py** extension and running it.

Steps to Write and Run a Program

- 1 Open a code editor (IDLE, VS Code, PyCharm).
- 2 Write the Python code.
- 3 Save the file with **.py** extension (e.g., `first_program.py`).
- 4 Run the program (Execute / Run the file).
- 5 View the output in the console.

Example - First Program

```
# This is our first Python program.  
print ("Hello, World!")  
# This line prints a message on the screen.
```

Output

Hello, World!

Explanation

- The **print()** function is used to display output.
- Text inside quotes (" ") is called a **string**.
- When we run the program, the message **Hello, World!** is displayed on the screen.

Different Ways to Print Text

Code	Output
<code>print("Hello")</code>	Hello
<code>print('Python is fun')</code>	Python is fun
<code>print("12345")</code>	12345
<code>print("Python", "Programming")</code>	Python Programming
<code>print("Hello\nWorld")</code>	Hello World

Important Points

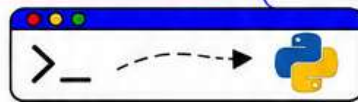
- Python is **case-sensitive**.
- Each statement is written in a **new line**.
- **Indentation** is important in Python.
- Use **meaningful** names for your files.
- Always save files with **.py** extension.

Common Errors (For Beginners)

- Forgetting quotes in print statement **X** `print(Hello)`
- Syntax error due to spelling mistakes **X** `prit("Hello")`
- Missing parenthesis **X** `print "Hello"`

Summary

- A program is a **set of instructions**.
- We use **print()** to display output.
- Save the file with **.py** extension.
- Run the program and view the output.
- **Practice** more programs to get better.



1.10.2 Understanding the print() Statement



* The `print()` function is used to display output on the screen.

Syntax

* It is the most commonly used function in Python.

```
print( value(s), sep = ' ', end = '\n')
```

* It prints the specified message or value.

Where,

- `value(s)` : The value(s) to be printed.
- `sep` : Separator between values (default is a space).
- `end` : Character added at the end (default is newline).

Examples



Code	Output
<code>print("Hello, World!")</code>	Hello, World!
<code>print(10)</code>	10
<code>print(3.14)</code>	3.14
<code>print(True)</code>	True
<code>print("Python")</code>	Python
<code>print("A", "B", "C")</code>	A B C

Using end Parameter



Code	Output
<code>print("Hello")</code>	Hello
<code>print("Hello", end='') print("World")</code>	HelloWorld
<code>print("Hello", end=' ') print("World")</code>	Hello World
<code>print("A") print("B")</code>	A B
<code>print("A", end=' ') print("B", end=' ') print("C")</code>	A B C

Using sep Parameter

Code	Output
<code>print("A", "B", "C")</code>	A B C
<code>print("A", "B", "C", sep=',')</code>	A,B,C
<code>print("1", "2", "3", sep='-')</code>	1-2-3

Escape Sequences in print()



Sequence	Meaning	Example Output
<code>\n</code>	New line	Hello World
<code>\t</code>	Tab space	A B
<code>\\</code>	Backslash	C:\Python
<code>\"</code>	Double quote	He said "Hi"
<code>\'</code>	Single quote	It's Python

Example Program



```
print("Python is fun!")  
print("Learning Python is easy.")  
name = "Easy2Learn"  
print("Welcome to", name)  
print(1, 2, 3, sep='-', end='!!\n')  
print("Done")
```

output
Python is fun!
Learning Python is easy.
Welcome to Easy2Learn
1 - 2 - 3 !!
Done

Key Points

- `print()` is used to display text, numbers or variables.
- Default separator is space and default end is newline.
- We can use `sep` and `end` to customize output.
- Escape sequences help in formatting the output.



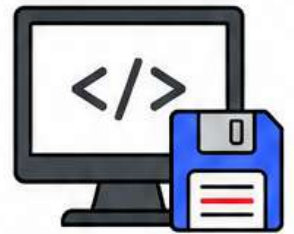
Summary

- `print()` is used to display output.
- Programs are saved with `.py` extension.
- Use meaningful names and save programs in a dedicated folder.

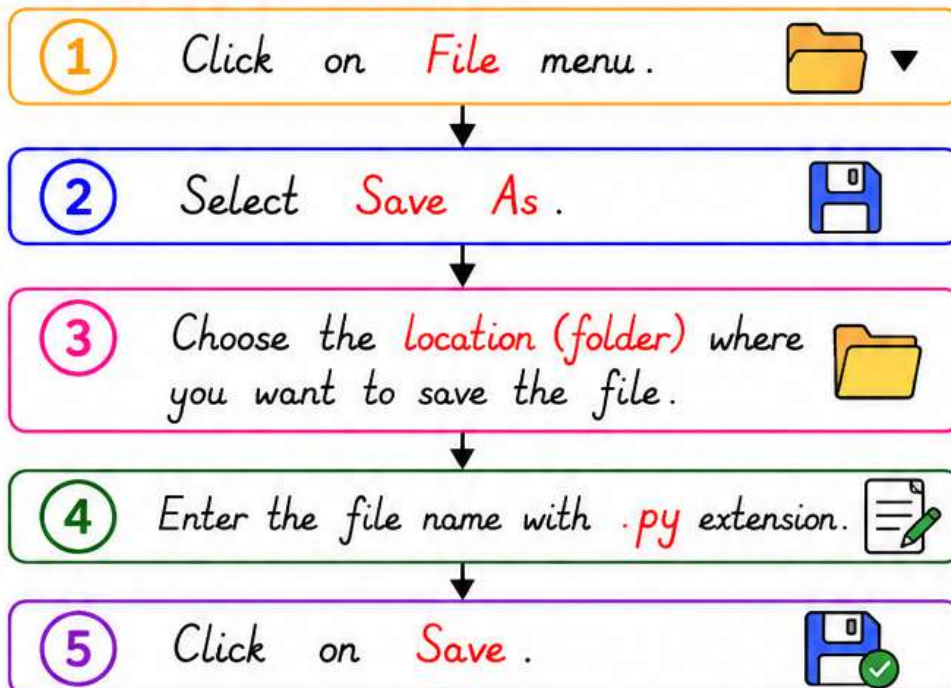


1.10.3 Saving the Program

- * After writing the code, we must **save the program** so that we can use it later.
- * Python programs are saved with **.py** extension.
- * A proper file name helps us to identify the program easily.



Steps to Save a Program



Example

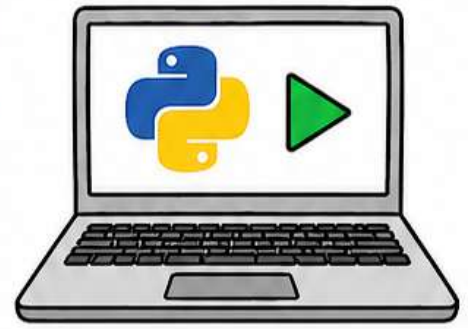
	File name	:	first_program.py
	Location	:	D:\Python_Programs
	Extension	:	.py

Points to Remember

- Always **save** your program before running.
- Use **meaningful names** for your files.
- Python files must have **.py** extension.
- Keep all your programs in a **dedicated folder**.



1.10.4 Executing the Program



- * Once we write a Python program, we need to **execute it** so that Python interpreter can run the code.
- * Python programs can be executed in **different** ways.

1. Ways to Execute a Python Program

① Using Python IDLE (Built-in)

Write the code and click on Run → Run Module.

② Using Command Prompt / Terminal

Navigate to the file location and use the command: **python filename.py**

③ Using an IDE (VS Code, PyCharm, etc.)

Click the Run button or press the shortcut key (usually F5).

Example

```
# program: hello.py  
print("Hello, World!")
```

Output

Hello, World!

2. Steps to Execute using Command Prompt

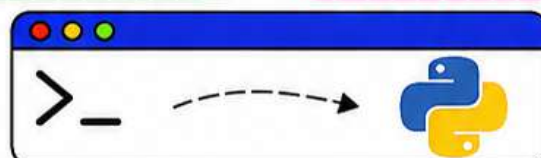


3. Important Points

- Make sure Python is installed and added to system **PATH**.
- The file name should have **.py** extension.
- If there is an error, Python shows the **error message** in the console.
- Save the file before executing.

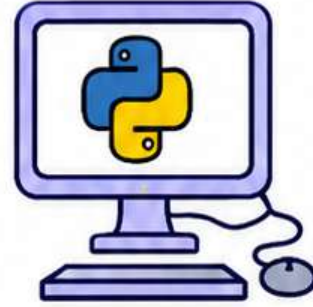
4. Summary

- Write the code in a **.py** file.
- Choose any one method to execute.
- Python runs the code line by line.
- The result (**output**) is shown on the screen.



1.10.5 Understanding the Output

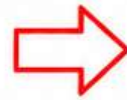
- * The **output** is the result displayed on the screen after executing a program.
- * It shows the values, text or results produced by the **print()** function.
- * Understanding output helps us **verify** and **debug** our programs.



Example Program

Code

```
# This program prints a message  
print("Hello, Python!")
```



Output (on screen)

Hello, Python!

Common Outputs and Their Meaning

Code	Output	Meaning
<code>print(10)</code>	10	Displays number 10.
<code>print(3.5)</code>	3.5	Displays decimal value.
<code>print("Python")</code>	Python	Displays text (string).
<code>print(2 + 3)</code>	5	Displays result of expression.
<code>print('A', 'B', 'C')</code>	A B C	Displays multiple values separated by space .

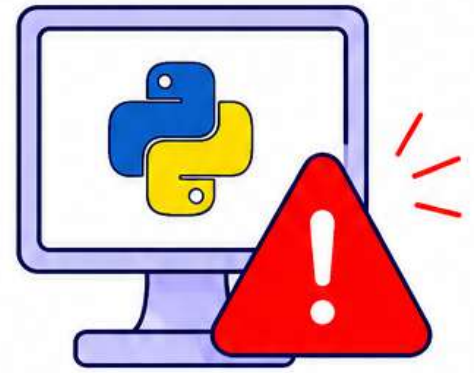
Key Points

- Output comes from using the **print()** function.
- It is shown in the **Python Shell** (console).
- Check the output to **confirm** your program is working correctly.
- Use output to **find** and **fix** errors in your code.



1.11 Common Beginner Errors

- * While writing Python programs, beginners may make small mistakes.
- * These errors stop the program from running correctly.
- * Learning and fixing them helps us write better code.



1.11.1 Syntax Error

- * A **syntax error** occurs when we do not follow the correct Python syntax (rules).
- * Python cannot understand the code and shows an error message.



Remember

Even a small mistake like a missing colon or bracket can cause a syntax error.

Common Examples

Mistake in Code	Error	Correct Code
<code>print("Hello")</code>	SyntaxError : '(' was never closed	<code>print("Hello")</code>
<code>if x > 0</code>	SyntaxError : expected ':'	<code>if x > 0:</code>
<code>print "Hello"</code>	SyntaxError : Missing parentheses	<code>print("Hello")</code>
<code>x = 5 +</code>	SyntaxError : invalid syntax	<code>x = 5 + 2</code>

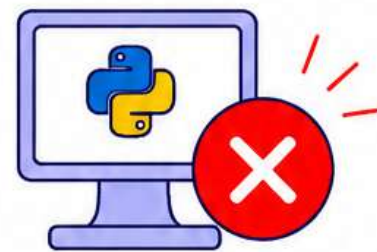
How to Avoid Syntax Errors

- Always read the **error message** carefully.
- Check **brackets, quotes, colons** and indentation.
- Use proper **Python syntax** and rules.
- Test and run the code regularly.



1.11.2 Indentation Errors

- * Python uses **indentation** (spaces) to define blocks of code.
- * Wrong indentation leads to **indentation errors**.
- * Use consistent indentation (usually **4 spaces**).



Common Examples

Mistake in Code	Error	Correct Code
<pre>if x > 0: print("Positive") print("Done")</pre>	IndentationError: expected an indented block	<pre>if x > 0: print("Positive") print("Done")</pre>
<pre>if x > 0: print("Positive") print("Done")</pre>	IndentationError: unexpected indent	<pre>if x > 0: print("Positive") print("Done")</pre>

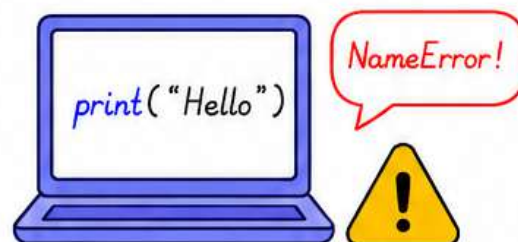
How to Avoid

- Always use **consistent indentation**.
- Recommended: **4 spaces** per indentation level.
- Do not mix **tabs** and **spaces**.
- Use an editor that shows indentation clearly.



1.11.3 Spelling and Case Errors

- * Python is **case-sensitive**.
- * Spelling mistakes in keywords, function names or variables cause errors.
- * Even a small mistake can stop the program.



Common Examples

Mistake in Code	Error	Correct Code
<pre>Print("Hello")</pre>	NameError: name 'Print' is not defined	<pre>print("Hello")</pre>
<pre>prInt("Hello")</pre>	NameError: name 'prInt' is not defined	<pre>print("Hello")</pre>
<pre>IF x > 0:</pre>	SyntaxError: invalid syntax	<pre>if x > 0:</pre>

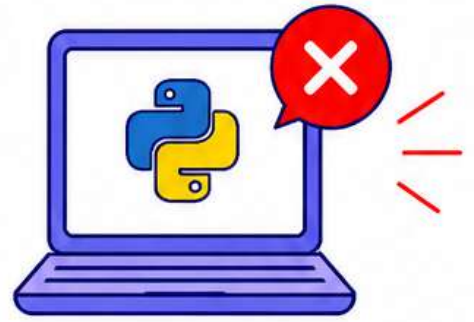
How to Avoid

- Check **spelling** of all keywords and function names.
- Python keywords are **lowercase**.
- Use a good IDE or editor with **auto-suggestions**.
- Read the error message carefully and correct it.



1.11.4 Missing Symbols

- * Missing symbols like **parentheses ()**, **colon :**, **quotes " "** or operators can cause errors.
- * Python expects certain symbols at the right place.
- * Always check for all required symbols.

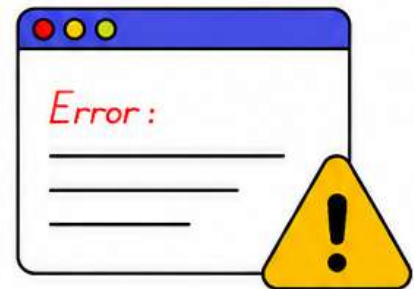


Common Examples

Mistake in Code	Error	Correct Code
<code>print "Hello"</code> (missing parentheses)	SyntaxError: Missing parentheses in call	<code>print("Hello")</code>
<code>if x > 0</code> (missing colon)	SyntaxError: expected ':'	<code>if x > 0:</code>
<code>print(Hello)</code> (missing quotes)	NameError: name 'Hello' is not defined	<code>print("Hello")</code>
<code>x = 5 + 2*</code> (missing operand)	SyntaxError: invalid syntax	<code>x = 5 + 2</code>

1.11.5 Reading Error Messages

- * Error messages help us **find** and **fix** problems.
- * They show **what** went wrong and **where**.
- * Read error messages carefully.
- * Use them to correct your code.



Parts of an Error Message

<code>File "script.py", line 3</code>	→	Name of the file and line number
<code>print("Hello")</code>	→	The line of code causing the error
<code>SyntaxError: invalid syntax</code>	→	Type of error and description

Tips

- Read the **full message**, not just the last line.
- Check the **line number** in your code.
- Understand the **error type** (SyntaxError, NameError, etc.).
- Fix the issue and **run the program again**.



1.12 Chapter Summary

This chapter introduced the basics of Python programming. Here is a quick summary of what we learned.



- 1.1 Introduction to Programming – What programming is and how computers **solve problems**.
- 1.2 Introduction to Python – What Python is, its history, **features**, advantages and limitations.
- 1.3 Applications of Python – Used in many fields like **web development, data science, AI, automation, robotics** and more.
- 1.4 Python and Other Languages – Compared Python with C, C++, Java and understood **interpreters**.
- 1.5 Python Program Execution – Source code → Bytecode → Python Virtual Machine → **Output**.
- 1.6 Python Versions – Major versions: Python 2 and **Python 3**. Python 3 is the latest and recommended.
- 1.7 to 1.9 Basics of Python – Program **structure, keywords, identifiers, comments** and documentation.
- 1.10 First Python Program – Write, save, run and understand the output using **print()** function.
- 1.11 Common Beginner Errors – **Syntax, indentation, spelling, missing symbols** and reading error messages.

★ Key Takeaway

Python is a powerful, easy and **versatile** programming language. It is widely used in **real-world** applications and is **beginner friendly**.

What We Achieved

- ✓ Understood the **meaning** and **importance** of programming.
- ✓ Learned about Python language and its **applications**.
- ✓ Understood how Python programs are **executed**.
- ✓ Wrote and executed the **first Python program**.
- ✓ Learned common errors and how to **avoid** them.



Remember

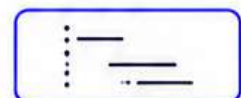
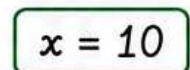
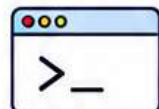
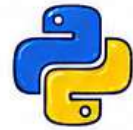
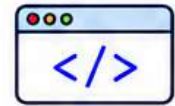
Practice regularly, make small programs and learn by doing. The more you code, the better you become!



1.13 Important Terms

Here are some important terms used in Python programming.

- **1. Program** :- A set of **instructions** written to perform a specific task.
- **2. Programming Language** :- A language used to write programs.
- **3. Python** :- A high-level, **interpreted** programming language.
- **4. Interpreter** :- A program that reads and executes code **line by line**.
- **5. Source Code** :- The **original code** written by a programmer.
- **6. Bytecode** :- **Intermediate code** generated by Python.
- **7. PVM (Python Virtual Machine)** :- **Executes** bytecode and produces output.
- **8. Statement** :- A single **instruction** in a program.
- **9. Identifier** :- A name given to **variables**, functions, classes, etc.
- **10. Keyword** :- **Reserved words** with special meaning.
- **11. Comment** :- **Notes** added to code for better understanding. (Ignored by Python)
- **12. Indentation** :- Spaces or tabs used to define **blocks of code**.
- **13. Output** :- The **result** produced after program execution.



Remember

Understanding these terms is the first step towards becoming a **good Python programmer**!

1.14 Review Questions

- * Answer the following questions to test your understanding of the concepts covered in this chapter.
- These questions are designed for **B.Tech** students and focus on **programming fundamentals**, **Python concepts**, **program execution**, **coding standards**, and **real-world applications**.

Part A - Very Short Answer Questions

- ① What is programming?
- ② What is a programming language?
- ③ Define Python.
- ④ Who developed Python?
- ⑤ What is meant by **source code**?
- ⑥ What is **bytecode**?
- ⑦ What is the **Python Virtual Machine (PVM)**?
- ⑧ What is an **identifier** in Python?
- ⑨ What is a Python **keyword**?
- ⑩ What is **indentation** in Python?

Part B - Short Answer Questions

- ① Differentiate between a program and a programming language.
- ② Explain the importance of problem-solving in programming.
- ③ List any **four** features of Python.
- ④ State any **four** advantages of Python.
- ⑤ Mention any **four** real-world applications of Python.
- ⑥ Differentiate between **Python 2** and **Python 3**.
- ⑦ Explain the role of the **Python Virtual Machine**.
- ⑧ What are **comments** in Python? Explain their importance.
- ⑨ Differentiate between Python **keywords** and **identifiers**.
- ⑩ List any **four** common beginner errors in Python.

TIP : Practice more questions and write code regularly to improve your understanding and coding skills. *

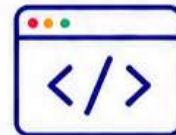


Part C – Descriptive Questions



- 1 Explain the concept of programming and its importance in software development.
- 2 Describe the history and evolution of Python.
- 3 Explain the major features of Python with suitable examples.
- 4 Discuss the advantages and limitations of Python.
- 5 Explain the various application areas of Python.
- 6 Compare Python with the C programming language.
- 7 Explain the execution process of a Python program from source code to output.
- 8 Describe the structure of a Python program with an example.
- 9 Explain Python keywords, identifiers, and naming conventions with examples.
- 10 Discuss comments, docstrings, and the importance of program documentation.

Part D – Programming Questions



- 1 Write a Python program to display "Hello, World!".
- 2 Write a Python program to print your name, branch, and college.
- 3 Write a Python program to display multiple lines using the `print()` function.
- 4 Write a Python program demonstrating the use of single-line comments.
- 5 Write a Python program using a `docstring`.
- 6 Write a Python program to print different data types using the `print()` function.
- 7 Write a Python program demonstrating correct indentation using an `if` statement.
- 8 Write a Python program to display the Python version installed on your system.
- 9 Write a Python program that prints formatted output using escape sequences (`\n`, `\t`).
- 10 Identify and correct the syntax errors in the given Python program.



★ Practice regularly, write **clean code**, and test your programs to improve your problem-solving and **coding skills**!



Part E - Higher-Order Thinking Questions (HOTS)



- ① Why is Python considered one of the most beginner-friendly programming languages?
- ② Explain why Python is widely preferred in Artificial Intelligence and Data Science compared to many traditional programming languages.
- ③ How does Python's interpreted nature influence program development and debugging?
- ④ Compare the advantages of using Python over C, C++, and Java for rapid application development.
- ⑤ Why is proper indentation considered both a syntax rule and a readability feature in Python?
- ⑥ How do comments and documentation improve software maintenance in large-scale projects?
- ⑦ Explain how Python's platform independence benefits software developers.
- ⑧ Analyze the complete execution flow of a Python program from writing the source code to generating the final output.
- ⑨ Discuss how coding conventions contribute to collaborative software development.
- ⑩ Suppose you are developing an engineering application. Justify why Python would be an appropriate choice by considering its features, libraries, and ease of development.

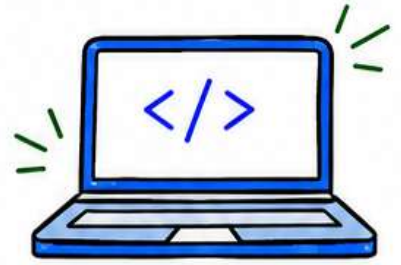


TIP : Think deeply, analyze **logically**, and express your answers clearly to strengthen your understanding and **problem-solving** abilities!



1.15 Solved Programs

* The following solved programs introduce the basic concepts of Python programming covered in this chapter. These examples help students understand Python syntax, the `print()` function, comments, indentation, escape sequences, and basic program structure.



Program 1: Hello World Program

Program

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

Output

Hello, World!

Explanation

- `print()` is used to display output on the screen.
- This is the first program written by most Python beginners.

Program 2: Print Student Details

Program

```
print("Name : Rahul")
print("Branch : Computer Science")
print("College : ABC Engineering College")
```

Output

Name : Rahul
Branch : Computer Science
College : ABC Engineering College

Explanation

- Multiple `print()` statements display information line by line.

Program 3: Printing Multiple Lines

Program

```
print("Python")
print("Programming")
print("Language")
```

Output

Python
Programming
Language

Explanation

- Each `print()` statement automatically moves to the next line.

Program 4: Using Escape Sequences

Program

```
print("Python\nProgramming")
print("Roll No\t101")
```

Output

Python
Programming
Roll No 101

Explanation

- `\n` → New line
- `\t` → Horizontal tab

Program 5: Using Single-Line Comments

Program

```
# This is a single-line comment
print("Welcome to Python")
```

Output

Welcome to Python

Explanation

- Comments improve code readability and are ignored by Python.

Program 6: Using Docstrings

Program

```
"""
This program displays
a welcome message.
"""
print("Python Programming")
```

Output

Python Programming

Explanation

- Docstrings provide documentation for modules, functions, and classes.

Program 7: Demonstrating Indentation

```
num = 10
if num > 0:
    print("Positive Number")
```

Output

Positive Number

Explanation

- Python uses indentation to define code blocks.



TIP : Practice these programs **regularly** to build **strong** Python programming skills.



Program 8: Checking Case Sensitivity

Program

```
value = 100
Value = 200
print(value)
print(Value)
```

Output

```
100
200
```

Explanation

- Python treats `value` and `Value` as different identifiers.

Program 9: Displaying Different Data Types

Program

```
print(100)
print(25.75)
print(True)
print("Python")
```

Output

```
100
25.75
True
Python
```

Explanation

- Python can display integers, floating-point numbers, Boolean values, and strings.

Program 10: Printing Special Characters

Program

```
print("\ Python \"")
print("C:\\ Python")
```

Output

```
"Python"
C:\ Python
```

Explanation

- `\\` prints double quotation marks.
- `\\` prints a backslash.

Program 11: Displaying Mathematical Expressions

Program

```
print("Sum = ", 20 + 30)
print("Product = ", 10 * 5)
```

Output

```
Sum = 50
Product = 50
```

Explanation

- Python evaluates arithmetic expressions inside the `print()` function.

Program 12: Using Multiple Arguments in print()

Program

```
name = "Anjali"
age = 20
print("Name:", name)
print("Age:", age)
```

Output

```
Name: Anjali
Age: 20
```

Explanation

- The `print()` function can display multiple values separated by commas.

Program 13: Displaying Python Keywords

Program

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

Sample Output

```
['False', 'None', 'True', 'and',
'as', 'assert', ...]
```

Explanation

- The `keyword` module provides the list of reserved Python keywords.

Program 14: Displaying Python Version

Program

```
sys
print(sys.version)
```

Sample Output

```
3.13.2 (main, ...)
```

Explanation

- The `sys.version` attribute displays the installed Python version.

Program 15: Program Showing Proper Formatting

Program

```
print("=" * 30)
print(" PYTHON PROGRAM")
print("=" * 30)
print("Name : Rahul")
print("Course : B.Tech")
print("Branch : CSE")
print("=" * 30)
```

Output

```
=====
PYTHON PROGRAM
=====
Name : Rahul
Course : B.Tech
Branch : CSE
=====
```

Explanation

- This program demonstrates formatted console output using repeated characters and aligned text.



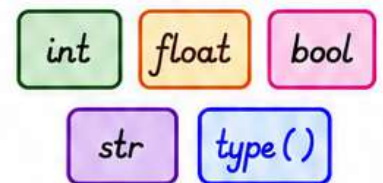
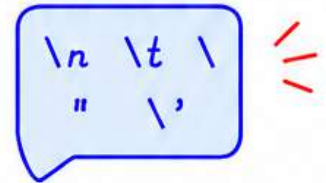
TIP: Practice these programs **regularly** to improve your **skills!**



1.16 Practice Programs

Practice Programs :

- 1 Write a Python program to display a welcome message, your name, branch, and college using the `print()` function.
- 2 Write a Python program to demonstrate the use of the `print()` function with multiple arguments, the `sep` parameter, and the `end` parameter.
- 3 Write a Python program to display formatted output using escape sequences such as `\n`, `\t`, `\\`, `"`, and `'`.
- 4 Write a Python program to demonstrate the use of single-line `comments` and a multiline `docstring` for program documentation.
- 5 Write a Python program to display different data types such as `int`, `float`, `bool`, and `str`, and identify their types using the `type()` function.
- 6 Write a Python program to demonstrate Python's case sensitivity by declaring variables with the names `value` and `Value` and displaying their values.
- 7 Write a Python program to display all Python keywords using the `keyword` module and count the total number of keywords.
- 8 Write a Python program to display the installed Python version, platform name, and interpreter information using the `sys` and `platform` modules.
- 9 Write a Python program to demonstrate `valid` and `invalid` identifiers by declaring valid variable names and commenting out invalid ones with appropriate explanations.
- 10 Write a Python program to identify and correct common beginner errors, including `syntax errors`, `indentation errors`, `spelling mistakes`, `case-sensitivity issues`, and `invalid symbols`, and explain them.



Valid Identifiers	Invalid Identifiers
✓ name	✗ 2name
✓ _name	✗ class
✓ name123	✗ name-1
✓ N1	✗ my name

