History of Python

1. **Origin (Late 1980s):**
   - Python was conceived in **December 1989** by **Guido van Rossum** at Centrum Wiskunde & Informatica (CWI) in the Netherlands.
   - Guido wanted to create a language that was easy to read, simple to use, but also powerful enough for system-level tasks.
2. **First Release (1991):**
   - Python 0.9.0 was released in **February 1991**.
   - It already included features like: exception handling, functions, and the core data types (str, list, dict).
3. **Python 1.x (1991–2000):**
   - Stable release of **Python 1.0** in **January 1994**.
   - Introduced modules, lambda functions, map, filter, and reduce.
4. **Python 2.x (2000–2010s):**
   - **Python 2.0** released in **October 2000**.
   - Introduced list comprehensions, garbage collection, Unicode support.
   - But backward incompatibility became an issue — many projects stuck on Python 2.
   - Final version: **Python 2.7 (2010)**, support officially ended **January 1, 2020**.
5. **Python 3.x (2008–Present):**
   - **Python 3.0** released in **December 2008**.
   - Major redesign: print became a function, integer division changes, better Unicode handling.
   - Initially, migration from Python 2 was slow, but today almost all projects use Python 3.
   - Current stable releases are in the **Python 3.11 – 3.13** range (fast, memory-efficient, type hints, async features).
6. **Community & Growth:**
   - Python is open-source and managed by the **Python Software Foundation (PSF)**.
   - Guido van Rossum was the "Benevolent Dictator For Life (BDFL)" until 2018, when he stepped down.
   - Today, Python is one of the **most popular languages** for data science, AI/ML, web development, automation, and education.

Features of Python

1. **Simple and Easy to Learn**
   - Python's syntax is clean and close to English, making it beginner-friendly.
   - Example: `print("Hello World")` instead of complex code.
2. **Interpreted Language**
   - No need for compilation. Python code runs directly using the interpreter.
   - Makes debugging and development faster.
3. **Dynamically Typed**
   - No need to declare variable types explicitly.
   - Example:
   - `x = 10     # integer`
   - `x = "Hi"  # now string, no error`
   -
4. **High-Level Language**
   - You don't need to manage memory or hardware-level details.
   - Python handles garbage collection automatically.

5. **Object-Oriented**
    - Supports **OOP concepts** like classes, inheritance, polymorphism.
    - Example:
    - ```
      class Car:
          def __init__(self, brand):
              self.brand = brand
      ```
6. **Extensive Standard Library**
    - Comes with a huge set of modules for math, file I/O, networking, regex, etc.
    - Example: `import math, datetime, os`
7. **Cross-Platform**
    - Python runs on Windows, Linux, macOS, and even embedded devices (Raspberry Pi).
8. **Open-Source & Community Driven**
    - Python is free to use and modify.
    - Backed by a strong global community and the PSF.
9. **Embeddable & Extensible**
    - Python can be integrated into C/C++ programs (embedding).
    - You can also call C/C++ code from Python (extending).
10. **Supports Multiple Paradigms**
    - Procedural (functions), Object-Oriented, Functional (map, reduce, lambda).
    - Example:
    - ```
      # functional
      nums = [1,2,3,4]
      print(list(map(lambda x: x**2, nums)))
      ```
11. **Portable**
    - "Write once, run anywhere." The same Python code runs on different OS without modification.
12. **Vast Ecosystem of Libraries & Frameworks**
    - For Web: Django, Flask, FastAPI
    - For Data Science: NumPy, Pandas, Matplotlib
    - For AI/ML: TensorFlow, PyTorch, Scikit-learn
    - For Automation: Selenium, PyAutoGUI
13. **Scalable & Widely Used**
    - Adopted by companies like Google, Facebook, Netflix, NASA.
    - Powers real-world applications from web apps to AI-driven systems.

Applications of Python

Python is widely used across many domains:

1. **Web Development**
    - Frameworks: Django, Flask, FastAPI
    - Example: Instagram, YouTube backend
2. **Data Science & Analytics**
    - Libraries: NumPy, Pandas, Matplotlib, Scikit-learn
    - Applications: Data cleaning, visualization, statistical analysis
3. **Artificial Intelligence & Machine Learning**
    - Libraries: TensorFlow, PyTorch, Keras, OpenCV

    - Applications: Chatbots, Recommendation systems, Computer vision
4. **Automation / Scripting**
    - Writing scripts for repetitive tasks
    - Example: File renaming, data scraping, testing
5. **Game Development**
    - Libraries: Pygame, Panda3D
    - Example: Prototyping games and simulations
6. **Embedded Systems / IoT**
    - MicroPython, CircuitPython
    - Example: IoT devices, robotics
7. **Scientific Computing**
    - Libraries: SciPy, SymPy, Jupyter
    - Applications: Research, simulations, mathematical modeling

Python Distributions

Python has multiple **distributions**, each tailored for specific use cases, platforms, or types of users. A Python distribution is essentially a packaged version of Python with certain libraries, tools, or environments pre-configured, which makes it easier to get started with development. Here's a detailed discussion:

1. Standard Python (CPython)

- **Description:** The default and most widely used implementation of Python, written in C.
- **Website:** python.org
- **Features:**
    - Maintained by Python Software Foundation (PSF).
    - Includes standard libraries and `pip` for package management.
- **Use Case:** General-purpose programming.
- **Example:** Installing Python from python.org and running `python3` on Linux or Windows.

2. Anaconda / Miniconda

- **Description:** A Python distribution focused on **data science, machine learning, and scientific computing**.
- **Website:** anaconda.com
- **Features:**
    - Comes with hundreds of pre-installed packages like `numpy`, `pandas`, `matplotlib`, `scikit-learn`.
    - Conda package manager for easy installation and environment management.
    - Miniconda is a lightweight version with just Python and Conda.
- **Use Case:** Data analytics, AI/ML, research.
- **Example:** `conda create -n myenv python=3.11`

3. ActivePython

- **Description:** A commercial Python distribution provided by ActiveState.
- **Website:** activestate.com
- **Features:**
    - Comes with precompiled libraries for enterprise applications.
    - Includes tools for security, package management, and performance.
- **Use Case:** Enterprise and commercial applications.

- **Example:** Suitable for organizations that need stable, supported Python builds.

4. WinPython

- **Description:** A portable Python distribution for **Windows**.
- **Website:** winpython.github.io
- **Features:**
  - No installation needed; can run from USB drive.
  - Includes scientific libraries and IDEs like Spyder.
- **Use Case:** Education, scientific work on Windows without admin privileges.
- **Example:** Just download, extract, and run Python directly.

5. PyPy

- **Description:** An alternative Python implementation focusing on **speed**.
- **Website:** pypy.org
- **Features:**
  - Uses Just-In-Time (JIT) compilation to make Python code run faster.
  - Compatible with most CPython code.
- **Use Case:** Performance-critical applications.
- **Example:** Running long loops or computation-heavy tasks faster.

6. MicroPython

- **Description:** Python for **microcontrollers and embedded systems**.
- **Website:** micropython.org
- **Features:**
  - Very lightweight and can run on small devices like ESP8266, ESP32.
  - Subset of Python standard libraries.
- **Use Case:** IoT projects and embedded programming.
- **Example:** Controlling LEDs, sensors using Python scripts on microcontrollers.

7. Jython

- **Description:** Python implementation that runs on the **Java Virtual Machine (JVM)**.
- **Website:** jython.org
- **Features:**
  - Integrates Python with Java libraries.
  - Allows using Python to call Java classes directly.
- **Use Case:** Projects requiring Python-Java interoperability.
- **Example:** Using Python to automate Java applications.

8. IronPython

- **Description:** Python for the **.NET framework**.
- **Website:** ironpython.net
- **Features:**
  - Fully integrates with .NET libraries.
  - Can be used in C# or VB.NET projects.
- **Use Case:** Windows applications using .NET ecosystem.
- **Example:** Python scripting within .NET applications.

9. Portable Python

- **Description:** Lightweight, portable Python distributions.

- **Features:**
  - Can run without installation.
  - Includes some preinstalled packages.
- **Use Case:** Quick testing or teaching Python in environments without installation rights.

Summary Table

Distribution	Target Platform / Use Case	Key Feature
CPython	General-purpose	Standard, official Python
Anaconda	Data Science, ML	Preinstalled scientific libraries
ActivePython	Enterprise	Supported, precompiled libraries
WinPython	Windows portable	No installation, scientific tools
PyPy	High performance	JIT compilation, faster execution
MicroPython	Embedded / IoT	Runs on microcontrollers
Jython	JVM integration	Python-Java interoperability
IronPython	.NET integration	Python-.NET interoperability
Portable Python	Lightweight, portable	No installation required

How Python Executes Code

When you run a Python script (`.py` file), Python internally does two things:

1. **Compilation to Bytecode**
   - Python converts the human-readable `.py` source code into **bytecode**, which is a lower-level, platform-independent representation of your code.
   - This bytecode is what Python actually executes.
   - Bytecode is stored in **`.pyc` files** (Python Compiled files).
2. **Interpretation**
   - The Python **interpreter** reads the bytecode and executes it on the Python Virtual Machine (PVM).
   - This is why Python is considered an interpreted language even though it has a compilation step.

What are `.pyc` Files?

- `.pyc` files are **compiled Python files** that contain **bytecode**.
- They are usually stored in a `__pycache__` folder in the same directory as your `.py` file.
- Example:
- `myscript.py → __pycache__/myscript.cpython-3.5.pyc`
-
- **Purpose**:
  - Speeds up execution because Python can skip the compilation step if the source hasn't changed.

- Makes distribution easier (though `.pyc` is not fully secure).

How `.pyc` Files Are Created

- Automatically: When you run a `.py` file:
- `python myscript.py`
-
- Python compiles it to bytecode and stores a `.pyc` file in `__pycache__`.
- **Manually: Using the `compileall` module:**
- **`python -m compileall myscript.py`**
- **`python3 -m compileall myscript.py`**
  - This will generate the `.pyc` file in the `__pycache__` folder.

How to Run `.pyc` Files

Once you have a `.pyc` file, you **don't need the original `.py` file** to run it. You can execute it directly:

```
python __pycache__/myscript.cpython-311.pyc
```

- Make sure to use the Python version that generated the bytecode.
- If you try to run a `.pyc` file generated by a different Python version, it might fail.

Coding Standards: PEP 8 guidelines

PEP8 is the **Python Enhancement Proposal** that provides guidelines for writing clean and readable Python code.

1. Indentation

Wrong:

```
def greet():
print("Hello")
```

Problem: No indentation after function definition.
Correct:

```
def greet():
    print("Hello")
```

Explanation: Python uses **4 spaces per indentation level**. This ensures readability and prevents syntax errors.

2. Maximum Line Length

Wrong:

```
print("This is a very long line that exceeds the recommended 79 characters in
PEP8 for better readability")
```

Correct:

```
print(
    "This is a very long line that exceeds the recommended 79 characters "
    "in PEP8 for better readability"
)
```

Explanation: Limit lines to **79 characters** for better readability and maintainability.

3. Blank Lines

Wrong:

```
def func1():
    pass
def func2():
    pass
```

Correct:

```
def func1():
    pass


def func2():
    pass
```

Explanation: Use **2 blank lines** before top-level functions or classes.

4. Imports

Wrong:

```
import sys, os
```

Correct:

```
import os
import sys
```

Explanation:

- One import per line
- Standard library imports first, then third-party, then local

5. Naming Conventions

Wrong:

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

MYvar = 10
```

Correct:

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

my_var = 10
```

Explanation:

- Functions/variables: `snake_case`
- Classes: `CapWords`
- Constants: `ALL_CAPS`

6. Spaces

Wrong:

```
x=10+5
my_list = [ 1,2, 3 ]
```

Correct:

```
x = 10 + 5
my_list = [1, 2, 3]
```

Explanation:

- Spaces around operators (+, =, etc.)
- No spaces inside brackets, commas followed by space

7. Comments and Docstrings

Wrong:

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

Correct:

```
def add(a, b):
    """Return the sum of two numbers."""
    return a + b
```

Explanation:

- Use **docstrings** for functions/classes
- Comments should be meaningful

Summary Table

Guideline	Wrong Example	Correct Example
Indentation	`print("Hello")`	`print("Hello")`
Max Line Length	Long single line	Split string into multiple lines
Blank Lines	No blank lines between funcs	2 blank lines between functions
Imports	`import sys, os`	`import os` `import sys`
Naming Conventions	`AddNumbers()`	`add_numbers()`
Spaces	`x=10+5`	`x = 10 + 5`
Comments/Docstrings	`#adds`	`"""Return the sum of two numbers."""`

Multiple Paradigms in Python

1. Procedural Programming (Imperative Style)

- Based on **procedures (functions)** and **step-by-step instructions**.
- Code executes line by line, using variables, loops, and conditions.
- Good for **small scripts, automation, and beginners**.

✅ **Example:**

```
# Procedural style
def area_of_circle(r):
    return 3.14 * r * r

radius = 5
print("Area:", area_of_circle(radius))
```

2. Object-Oriented Programming (OOP)

- Based on **objects (data + behavior)**.
- Uses **classes, inheritance, encapsulation, polymorphism**.
- Good for **large projects, GUI apps, simulations, real-world modeling**.

✅ **Example:**

```
# Object-Oriented style
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

c = Circle(5)
print("Area:", c.area())
```

3. Functional Programming

- Treats functions as **first-class citizens** (can be passed as arguments, returned as values).
- Emphasizes **immutability, recursion, and higher-order functions**.
- Good for **data processing, mathematical computations, concise code**.

✅ **Example:**

```
# Functional style
nums = [1, 2, 3, 4, 5]

# Using map + lambda
squares = list(map(lambda x: x**2, nums))

# Using filter
evens = list(filter(lambda x: x % 2 == 0, nums))

print("Squares:", squares)
print("Evens:", evens)
```

4. Scripting / Declarative Style

- Python can be used as a **scripting language** to automate tasks (file handling, system operations).
- Declarative style (like SQL) is possible using **libraries** (e.g., ORM in Django).

✅ **Example (Scripting):**

```
# Scripting style
import os

files = os.listdir(".")
print("Files in current directory:", files)
```

Summary Table

Paradigm	Key Idea	When to Use	Example
Procedural	Functions, step-by-step instructions	Small programs, simple tasks	`def func(): ...`
OOP	Classes & Objects (data + methods)	Large, complex projects	`class Car: ...`
Functional	Functions as first-class objects, immutability	Data science, math-heavy tasks	`map(), filter(), reduce()`
Scripting	Automating tasks with short scripts	File mgmt, automation, sys admin	`os.listdir()`