Practical - 1

Aim:

Introduce the basics of the Python language, including history, versions, and features.

Q1: History of Python

Ans:

Python was created by Guido van Rossum in the late 1980s (at CWI in the Netherlands) and first released in February 1991 (as version 0.9.1). Named after the Monty Python comedy troupe, Python was designed as a successor to the ABC language, emphasizing readable syntax and ease of use. Its development was led by van Rossum (the "Benevolent Dictator for Life" until 2018), and the language has since grown through community contributions. Over time, Python's scope expanded to include features like list comprehensions, garbage collection, and Unicode support in Python 2.0 (released October 2000), and a major backwards-incompatible redesign with Python 3.0 (released December 2008). Python 2.7.18 (April 2020) was the final release of the 2.x series. Today Python 3.x is the standard: as of late 2025 the latest stable release is Python 3.14.2 (Oct 2025), with older 3.x versions in security support.

Key milestones in Python's history include:

- **1991 – Python 0.9.1:** First public release (Feb 1991) including classes, exception handling, functions, and the core datatypes (list, dict, str)
- **1994 – Python 1.0:** Official version 1.0 released (Jan 1994), adding functional features (map, filter, reduce) and solidifying the module system
- **2000 – Python 2.0:** Released October 2000 with new features (list comprehensions, cycle-detecting garbage collector, Unicode support). The 2.x series continued up to 2.7.18 (April 2020)
- **2008 – Python 3.0:** Released December 2008 as a major overhaul (incompatible with 2.x) with improved Unicode, print function, and other changes. Python 3 has since become the focus of development.
- **2025 – Python 3.14:** The current stable series (Python 3.14.2 released Oct 2025). Python 3.10–3.14 are actively supported, while Python 3.15 is in development.

Q2: Differentiate compiler and interpreter

Ans:

A compiler and an interpreter are both language processors, but they work differently. A compiler translates an entire program's source code into machine code (often a binary executable) before execution. Once compiled, the program runs directly on the hardware, which often makes execution faster. However, compiling the whole program can take time and requires recompilation whenever the source code changes. In contrast, an interpreter translates and executes code line by line at runtime. This means Python source code is converted to machine actions on the fly. Because interpretation happens one statement at a time, development is more interactive: errors are caught immediately as each line runs, easing debugging. The tradeoff is that interpreted execution can be slower, since translation happens during execution.

- **Compiler:** Translates the entire program to machine code in one go, producing an executable. Results in faster runtime performance (since machine code runs natively), but requires a complete compile step and recompilation on changes. Examples of compiled languages include C, C++, and C#.
- **Interpreter:** Translates and executes code statement by statement at runtime. No separate compile step is needed. This allows quick testing and debugging (errors show up immediately), but each run may be slower. The Python interpreter (CPython) works this way, as do languages like Ruby and Perl.

Q3: What types of applications can be developed by using Python?

Ans:

Python's versatility makes it useful across many domains. Common application areas include:

- **Web & Internet Development:** Python has powerful web frameworks (e.g. Django, Flask, Pyramid) and CMS systems (Plone, django CMS). Its standard library and modules also support web protocols (HTML, XML, JSON, email, FTP, etc.).
- **Scientific & Numeric Computing:** Libraries like SciPy (for math and engineering), NumPy (numeric arrays), Pandas (data analysis), and IPython (enhanced interactive shell) make Python ideal for research and data work. Python's ecosystem (matplotlib, scikit-learn, TensorFlow, etc.) further enables machine learning, simulation, and statistics.

- **Education:** Python’s simple syntax and readability make it popular in teaching programming. Many introductory CS courses and resources use Python, and there is an Education Special Interest Group promoting its use.
- **Desktop GUI Applications:** Python supports desktop apps via toolkits like Tkinter (built-in), PyQt/PySide, wxPython, and Kivy (for touch/mobile). These let developers create cross-platform graphical interfaces.
- **Software Development & Scripting:** Python is often used as a “glue” language for build automation, testing, and tools. Examples include SCons (build tool), Buildbot or Apache Gump (CI/testing), and Trac/Roundup (project management/bug tracking).
- **Business & Enterprise:** Python powers many business applications, from simple automation scripts to full ERP systems. For instance, Odoo is a Python-based suite for enterprise management, and Tryton is a modular business app platform. These leverage Python’s rapid development and wide library support.

Each of these domains is well-supported by Python’s standard library and the large PyPI collection of third-party modules.

Q4: List down Python versions.

Ans:

Python has evolved through major version series. The key versions include:

- **Python 0.x and 1.x:** The initial implementation (0.9.1 in Feb 1991) introduced core features. Python 1.0 (Jan 1994) and subsequent 1.x releases (up to 1.6 in Sep 2000) added features like functional tools and improved I/O.
- **Python 2.x:** Python 2.0 (Oct 2000) brought significant enhancements (comprehensions, garbage collection, Unicode). The 2.x series continued (2.1, 2.2, …) with bugfixes and minor features through 2.7.18 (released Apr 2020). Python 2.7.18 is the final 2.x release; Python 2 is officially unsupported after 2020.
- **Python 3.x:** Python 3.0 (Dec 2008) was a major rewrite (not backward-compatible with 2.x), introducing features like the **print()** function, new division behavior, and improved Unicode. The 3.x series has been incrementally improved through versions 3.1, 3.2, … up to the current. As of late 2025, the latest stable version is Python 3.14.2 (Oct 2025). Python 3.10, 3.11, 3.12, 3.13, and 3.14 are supported; Python 3.9 and earlier are only in security-fix phase or end-of-life. Python 3’s development continues (with 3.15 and beyond), following a release cycle of roughly one major version per year.

Q5: List out Python IDE.

Ans:

Many development environments support Python. Common integrated development environments (IDEs) and code editors include:

- **IDLE:** The default Python IDE bundled with CPython. A simple editor with syntax highlighting and a debugger, suitable for beginners.
- **PyCharm:** A full-featured Python IDE (from JetBrains) popular with professional developers. It offers smart code navigation, refactoring, integrated testing, and framework support. (There is a free Community Edition and a paid Professional Edition.)
- **Visual Studio Code (VS Code):** A free, lightweight editor from Microsoft. With Python extensions installed, VS Code becomes a powerful IDE with autocomplete, debugging, and Git integration. It is widely used for Python and many other languages.
- **Jupyter Notebook/Lab:** An interactive notebook environment (originally IPython) extensively used in data science. Not technically a classic IDE, but it lets you write and execute Python code blocks with rich output (charts, math, etc.) in a document.
- **Spyder:** An open-source scientific IDE (bundled with Anaconda) that resembles MATLAB. It includes an editor, interactive console, variable explorer, and plotting integration, making it good for data analysis work.
- **Eclipse + PyDev:** Eclipse is a powerful multi-language IDE. With the PyDev plugin, it supports Python development (code completion, debugging).
- **Thonny:** A simple Python IDE designed for learning and beginners. It has a debugger and a clear view of variables, making it easy to step through code.
- **Others:** Developers also use editors like Sublime Text, Atom, Vim/Neovim, Emacs, or IDEs like Wing IDE, Komodo, Eric Python IDE, etc. (These can be customized with Python plugins.)

Each of these environments offers different trade-offs (simplicity vs. features), but all support writing and running Python code.

Q6: List out the features of Python.

Ans:

Python has many characteristic features that distinguish it from other languages. Key features include:

- **High-level, Readable Syntax:** Python is a high-level general-purpose language emphasizing code readability. It uses indentation (whitespace) instead of braces for blocks, and English-like keywords, making code concise and clear.
- **Dynamically Typed:** Variables in Python are not declared with a type; the type is determined at runtime. This allows rapid coding without boilerplate. (However, it means type errors may only appear during execution.)
- **Interpreted:** Python code is executed by an interpreter at runtime, translating line by line. This supports interactive programming and quick testing (though execution is typically slower than compiled languages).
- **Multi-paradigm:** Python fully supports object-oriented programming (classes and objects with inheritance, encapsulation, polymorphism) and also procedural and functional styles. This flexibility allows choosing the best approach for a problem.
- **Extensive Standard Library ("Batteries Included"):** Python comes with a rich standard library that includes modules for file I/O, web services, databases, math, text processing, and more. This means common tasks can be done without installing additional packages.
- **Open-Source & Portable:** Python is free and open-source (under the Python Software Foundation License). It runs on virtually all platforms (Windows, Linux/Unix, macOS, etc.) with minimal changes.
- **Automatic Memory Management:** Python has built-in garbage collection (reference counting with cycle detection). Programmers do not manually allocate or free memory for objects.
- **Large Ecosystem:** Beyond the standard library, Python has a vast ecosystem of third-party libraries and frameworks (for web, data science, AI, etc.) contributed by its active community.

These features (simplicity, rich libraries, portability, etc.) make Python suitable for rapid development of many kinds of programs.

Q7: Write advantages of Python language over other languages.

Ans:

Compared to many other programming languages, Python offers several advantages:

- **Easy to Learn & Use:** Python's clear, concise syntax is simpler than verbose languages like C++ or Java. This reduces complexity and learning time. Python code often requires fewer lines to express the same logic (e.g. a loop or conditional) compared to other languages.
- **Rapid Development:** Because of its high-level data types (lists, dicts, etc.) and interpreted nature, Python is excellent for quick prototyping. The interactive REPL (read–eval–print loop) lets developers test ideas immediately. Many startups and developers choose Python when speed of development is critical.
- **Extensive Libraries and Frameworks:** Python's ecosystem provides ready-made solutions for almost every need. Libraries for web development (Django, Flask), data science (NumPy, Pandas, TensorFlow), automation, and more let developers avoid "reinventing the wheel." This means common tasks can be implemented with minimal code. Few languages match Python's breadth of community-contributed packages.
- **Portability:** Python runs on all major platforms. A Python program written on one OS generally works on another without modification. This "write once, run anywhere" quality simplifies cross-platform deployment (unlike languages that compile to platform-specific binaries).
- **Strong Community and Industry Support:** Python has a large, active user community. Many problems are already solved in open-source projects. Moreover, major tech companies (Google, Facebook, Microsoft, etc.) support Python development. This corporate backing and community presence mean Python is continually improved and well-documented.
- **Versatility:** Python's multi-paradigm nature and extensive libraries make it suitable for web, desktop, scientific, and embedded applications alike. It integrates well with other systems (e.g. calling C/C++ code) and can "glue" together components written in different languages.

These factors give Python an edge for many projects: rapid iteration, ease of maintenance, and a rich toolset of libraries all help development be faster and less error-prone compared to more low-level or narrow-purpose languages.

Q8: Differentiate dynamic and static types programming.

Ans:

Programming languages can be classified by how they handle data types: **static typing vs dynamic typing:**

- **Static Typing:** In statically-typed languages (like C or Java), every variable has a fixed type known at compile time. The type must usually be declared (e.g. **int x;**), and the compiler checks type correctness before running the program. Errors such as assigning a string to an integer variable are caught during compilation. This can improve runtime performance and help catch type errors early.
- **Dynamic Typing:** Dynamically-typed languages (like Python, JavaScript, Ruby) perform type checking at runtime. Variables are just names bound to values, and those values carry the types. You do not declare types explicitly. For example, in Python you can write **num = 5** (an integer) and later **num = "hello"** (a string) without error at compile time. Type compatibility issues only appear when the code is executed. This gives more flexibility and faster coding (no need to specify types), but it means some errors may only surface during program execution.

In summary: static typing means “type errors catch before running” and usually requires explicit declarations; dynamic typing means “types are checked on the fly” and allows more flexible, succinct code at the cost of potentially later error detection.

Q9: Differentiate procedural and object oriented programming.

Ans:

Procedural and object-oriented programming are two fundamental paradigms:

- **Procedural Programming:** Organizes code into procedures or functions that operate on data. The program’s logic is a sequence of function calls and global data structures. It follows a top-down approach. Functions (procedures) contain a series of computational steps, and the same data may be passed through many functions. There is generally no built-in mechanism to encapsulate or hide data. Procedural code tends to treat data and functions as separate: “functions are more important than data”. Examples of procedural languages include C, Fortran, and Pascal.
- **Object-Oriented Programming (OOP):** Organizes code around objects, which are instances of classes. Each object bundles data (attributes) and behavior (methods) together. OOP follows a bottom-up approach: you define classes, then create objects from them. Key features include encapsulation (hiding an object’s internal state), inheritance (classes inheriting

attributes/methods from parent classes), and polymorphism. In OOP, data and methods form self-contained entities. The paradigm models real-world entities and relationships (objects interact with one another). Data can be marked private or public (data hiding), which enhances security and modularity. In OOP code, you typically say "data is more important than function" because behavior is tied to the objects themselves. Languages that support OOP include Python, Java, C++, C#, and many modern languages.

Key differences:

- **Structure:** Procedural divides a program into functions; OOP divides a program into classes/objects.
- **Data Handling:** Procedural code passes data between functions (no inherent data hiding). OOP encapsulates data within objects, allowing abstraction and hiding of internal state.
- **Approach:** Procedural is top-down (solve problem step by step); OOP is bottom-up (build objects that interact).
- **Features:** OOP supports inheritance and polymorphism (code reuse through class hierarchies), whereas pure procedural programming has no native inheritance.
- **Use Cases:** Procedural style can be simpler for small or straightforward tasks, while OOP scales better to complex systems by modeling real-world relationships.

In practice, many languages (like Python and C++) support both styles. You can write procedural code in Python, or use its full object-oriented features. The choice depends on the problem domain and design preferences.

Conclusion:

Python is a powerful and beginner-friendly programming language that allows developers to build applications quickly with clean and readable code. Its vast standard library, strong community support, and wide range of applications from web development to AI make it extremely versatile. Because Python reduces development time while remaining highly capable, it is an excellent choice for students, professionals, and researchers alike.