

By Pranit Dalvi

Introduction

This documentation covers various topics related to Python, coding standards, and APIs. It provides a brief overview of the key aspects of Python programming.

***Python***

*Python syntax, variables, datatypes (int, float, string, list, tuple, dict, set):*

The foundation of Python lies in its syntax, variable management, and data types. Being dynamically typed, Python infers variable types at runtime, allowing reassignment to different types without explicit declaration. While this flexibility reduces boilerplate, it also risks type-related bugs, making testing and documentation essential. The type() function helps inspect a variable’s type. Python integers have no size limit, enabling very large number computations.

Python provides primitive types such as int, float, str, and bool. For collections, it includes:

**List**: ordered, mutable, allows mixed data.

**Tuple**: ordered, immutable, hashable—usable as dictionary keys or set elements.

**Dictionary**: unordered key-value pairs; keys must be unique and immutable.

**Set**: unordered, mutable, no duplicates.

*Conditional statements (if-elif-else):*

**1. Conditional Statements**

**if statement**  
Definition: Executes a block of code only if the condition is True.  
Explanation: Used for simple decision-making.

**if-else statement**  
Definition: Provides two execution paths: one if the condition is True, another if it is False.  
Explanation: Ensures one of the two code blocks always runs.

**if-elif-else chain**  
Definition: Tests multiple conditions in sequence.  
Explanation: The first condition that evaluates to True is executed; others are skipped.

**Logical operators (and, or, not)**  
Definition: Operators that combine or invert conditions.  
Explanation: Allow building complex logical expressions.

*Loops (for, while, break, continue):*

**2. Loops**

**for loop**  
Definition: Iterates over the items of a sequence (list, tuple, string, range, etc.).  
Explanation: Used when the number of iterations is known or when working with collections.

**while loop**  
Definition: Executes a block repeatedly as long as the condition remains True.  
Explanation: Useful when the number of iterations is not predetermined.

**Infinite loop (while True)**  
Definition: A loop that runs indefinitely because the condition is always True.  
Explanation: Usually controlled with a break statement.

**Loop Control Statements**

**break**  
Definition: Immediately stops the nearest enclosing loop.  
Explanation: Used when a loop must end before completing all iterations.

**continue**  
Definition: Skips the rest of the current iteration and moves to the next one.  
Explanation: Helps skip unwanted cases without breaking the loop.

**else with loops**  
Definition: Runs only if the loop finishes without encountering a break statement.  
Explanation: Often used in search operations to avoid extra flag variables.

*Functions (parameters, return values, default args, \*args, \*\*kwargs)*

A function is a defined, reusable block of code that performs a specific action. When a function is called, it can be passed input values as

**parameters**. The return statement is used to send a value or object back to the caller and immediately terminates the function's execution. Python functions can return multiple values, which are packaged as a single

tuple object for convenience.

Python offers flexible parameter handling:

**Default arguments** have a pre-assigned value that is used if the caller does not provide one.

**\*args** allows a function to accept a variable number of non-keyword arguments, which are collected into a tuple.

**\*\*kwargs** allows a function to accept a variable number of keyword arguments, which are collected into a dictionary.

*Exception Handling (try-except-finally)*

try-except-finally blocks provide a robust way to manage these events. The try block contains the code that may raise an exception. The except block catches and handles a specific exception if one occurs in the try block. It is possible to have multiple except blocks for different types of exceptions. The optional else block is executed only if the code in the try block runs without any exceptions. The finally block is executed regardless of whether an exception occurred, was handled, or if a return statement was encountered in the try or except blocks. This makes it ideal for cleanup tasks like closing files or releasing resources.

*Decorators*

A decorator is a function that takes another function as an argument, extends its behaviour without permanently modifying it, and then returns the wrapped function. The underlying concept that makes this possible is that functions in Python are "first-class objects," meaning they can be passed as arguments and returned from other functions.

The @ symbol is a convenient syntactic shortcut for applying a decorator to a function definition. Python's standard library includes several built-in decorators, such as:

@classmethod: Converts a method into a class method, which receives the class itself (cls) as its first argument.

@staticmethod: Transforms a method into a static method that does not receive any implicit first argument (self or cls).

@property: Turns an instance method into a "getter" for a read-only attribute, often used in conjunction with a @.setter to control attribute modification.

*OOPS*

OOP is a programming paradigm that organizes code by bundling related data (attributes) and behaviours (methods) into individual objects. A **class** serves as a blueprint for creating objects, and a **class variable** is shared by all instances of the class.  An **object** is a unique instance of a class. The special   \_\_init\_\_() method, or constructor, is automatically called to initialize the object's unique state when it is created. An **instance variable** is unique to each object.

The four core principles of OOP are:

**Encapsulation:** The practice of bundling data and methods into a single unit (the class) and restricting external access to the internal state.

**Inheritance:** Allows a child class (subclass) to acquire the attributes and methods of a parent class (superclass), which promotes code reuse.

**Polymorphism:** Allows objects of different types to be treated as instances of a common base type, as long as they share a similar interface or behaviour. This is often achieved through method overriding, where a subclass redefines a method from its parent. It is also related to "duck typing," where an object can be treated as a specific type if it has the required methods, regardless of its actual class.

**Abstraction:** The process of hiding complex implementation details and exposing only the essential functionality to the user.

*List comprehension, dictionary comprehension*

List Comprehension

**Definition:** A compact way to generate a list from an iterable (like a range, list, or string) using a single line.

**Syntax:**

[expression for item in iterable if condition]

Dictionary Comprehension

**Definition:** A concise way to create dictionaries from iterables or other mappings.

**Syntax:**

{key\_expr: value\_expr for item in iterable if condition}

*Iterators & Generators*

When working with large datasets, it is often impractical to load all the data into memory at once. Iterators and generators solve this problem by processing data one item at a time.

An **iterable** is any object that can be looped over, such as a list or string.

An **iterator** is an object that implements the "iterator protocol" by defining two methods: \_\_iter\_\_() and \_\_next\_\_(). The

\_\_next\_\_() method returns the next item in the sequence and raises a StopIteration error when there are no more items.

A **generator** is a simpler and more memory-efficient way to create an iterator. A function becomes a generator when it uses the yield keyword instead of return. The yield statement pauses the function, returns a value, and saves its state, allowing it to resume from where it left off on the next call. This "lazy evaluation" prevents the generator from storing all its contents in memory at once.

***Coding Standards***

Python coding standards are primarily guided by PEP 8, the official style guide for Python code. Following these standards helps ensure your code is readable, consistent, and professional. Here's a breakdown of the key principles:

Core Style Guidelines (PEP 8)

* Indentation: Use 4 spaces per indentation level (no tabs).
* Maximum Line Length: Limit lines to 79 characters.
* Blank Lines: Use blank lines to separate functions, classes, and sections.
* Imports: Group imports in the order: standard libraries, third-party libraries, local application imports.
* Whitespace:
* Avoid extra spaces inside parentheses, brackets, or braces.
* Use a single space after commas and around binary operators.

*Naming Conventions*

* Variables and Functions: Use lower\_case\_with\_underscores.
* Classes: Use CapitalizedWords.
* Constants: Use ALL\_CAPS\_WITH\_UNDERSCORES.
* Private Members: Prefix with a single underscore \_.

*Comments and Docstring*

* Block Comments: Use complete sentences and indent at the same level as the code.
* Inline Comments: Use sparingly and separate from code with at least two spaces.
* Docstrings: Use triple quotes """ for module, class, and function documentation (PEP 257).

*Types of testing*

1. **Unit Testing**

* **Purpose**: Test individual functions or components in isolation.
* **Tools**: unittest, pytest
* **Example**: Testing a calculate\_risk\_score() function with known inputs.

def test\_calculate\_risk\_score(): assert calculate\_risk\_score(critical=True, age=5) == 90

2. **Integration Testing**

* **Purpose**: Test how multiple modules work together.
* **Tools**: pytest, tox
* **Example**: Testing Flask routes with database access and email sending.

3. **Functional Testing**

* **Purpose**: Validate that the system behaves as expected from the user's perspective.
* **Tools**: selenium, behave
* **Example**: Simulating a user logging in and submitting a form.

4. **Regression Testing**

* **Purpose**: Ensure new changes don’t break existing functionality.
* **Tools**: pytest, CI/CD pipelines (GitHub Actions, Jenkins)
* **Example**: Re-running all tests after adding a new feature to asset scoring.

5. **Mock Testing**

* **Purpose**: Simulate external systems or dependencies.
* **Tools**: unittest.mock, pytest-mock
* **Example**: Mocking SMTP server during email tests.

from unittest.mock import patch @patch('smtplib.SMTP') def test\_email\_sender(mock\_smtp): assert send\_email("test@example.com", "Subject", "Body") is True

6. **Smoke Testing**

* **Purpose**: Quick check to ensure basic functionality works.
* **Tools**: Manual or scripted tests
* **Example**: Launching the dashboard and verifying login, homepage, and asset list load correctly.

7. **End-to-End (E2E) Testing**

* **Purpose**: Test the entire application flow from start to finish.
* **Tools**: Playwright, Selenium
* **Example**: Simulating a user onboarding, adding assets, and generating compliance reports.

8. **Exploratory Testing**

* **Purpose**: Manual testing without predefined cases—used to discover unexpected issues.
* **Tools**: Human intuition + app usage
* **Example**: Clicking through your dashboard to find edge cases or UI bugs.

Want help writing unit tests for your Flask routes or Tkinter logic? I can walk you through it with real examples from your code.

*PEP 8 Highlights*

PEP 8 is the official Python style guide. Key rules:

* Indentation: 4 spaces per level.
* Line length: max 79 characters.
* Imports: standard → third-party → local.
* Whitespace: avoid extra spaces inside brackets.
* Use is/is not for None comparisons.
* Prefer f-strings over % or .format().

Use tools like black, flake8, or autopep8 to enforce it.

*SOLID and DRY Principles*

SOLID (Object-Oriented Design)

* **S**: Single Responsibility – each class/function does one thing.
* **O**: Open/Closed – open for extension, closed for modification.
* **L**: Liskov Substitution – subclasses should replace base classes without breaking behaviour.
* **I**: Interface Segregation – avoid forcing classes to implement unused methods.
* **D**: Dependency Inversion – depend on abstractions, not concrete implementations.

DRY (Don't Repeat Yourself)

* Avoid duplicating logic or code.
* Use functions, classes, and modules to reuse behaviour.
* Centralize configuration and constants.

***APIs***

*Types of APIs*

APIs, or Application Programming Interfaces, are software tools that allow two applications to communicate with each other. They are classified based on their usage patterns and architectures. Common types include:

* **RESTful APIs:** Use the HTTP protocol and are lightweight, making them ideal for web and mobile applications.
* **SOAP APIs:** A more rigid protocol that uses XML for its data format, often used in large enterprise systems.
* **Open APIs (or Public APIs):** These are publicly available to developers.
* **Internal APIs:** Used only by specific departments within a company and are hidden from external users.
* **Partner APIs:** Used to enable communication between the systems of a company and its business partners.

**HTTP Status codes**

HTTP status codes are standardized numerical codes that communicate the outcome of an HTTP request. They are grouped into five classes :

* **1xx (Informational):** The request has been received, and the process is continuing.
* **2xx (Success):** The request was successfully received, understood, and accepted.

200 OK is a common success code.

* **3xx (Redirection):** Further action is required to complete the request.
* **4xx (Client Error):** The request contains bad syntax or cannot be fulfilled.

404 Not Found and

401 Unauthorized are common examples.

* **5xx (Server Error):** The server failed to fulfill a valid request.

*Response Formats*

A consistent and standardized error response structure is a best practice for API design. Instead of just a status code, a detailed error response in a format like JSON can provide more context for debugging. It is also important to adopt a "deny all and selectively allow" approach to prevent data leakage in responses by using allowlisting for explicitly permitted fields.

*Types of API Auth*

API authentication is the process of verifying a client's identity to ensure that only authorized systems or users can interact with an API. Common methods include:

* **Basic Authentication:** The client sends a username and password in the request header.
* **API Key Authentication:** The API provider assigns a unique key to each client, which is then included in every request, often in the header or as a query parameter.
* **Token-Based Authentication:** Uses a dynamic, short-lived access token to verify a user's identity. This is a popular method that provides a higher level of security than API keys.

*Versioning and Security*

API versioning is necessary when a "breaking change" is introduced that would require clients to modify their codebase. Planning a versioning strategy early can save significant effort. Common versioning strategies include:

* **URL Path Versioning:** Embedding the version in the URL, such as /v1/products.
* **Query Parameter Versioning:** Adding the version as a query parameter, such as /products?version=1.
* **Header Versioning:** Using a custom HTTP header to specify the version.
* **Media Type Versioning:** Using the Accept header to specify the version, for example Accept: application/vnd.myapi.v2+json.

For security, it is critical to implement precise and consistent status codes to strengthen the API's defensive posture. You should also implement a security-conscious logging strategy that includes request metadata, response status codes, and error categories.

*CRUD operations*

CRUD stands for **C**reate, **R**ead, **U**pdate, and **D**elete. These are the four fundamental operations that a client can perform on data in a database. In a RESTful API, these operations typically correspond to specific HTTP methods :

* **Create:** Performed using a POST request, which is used to send data to a server to create a new resource.
* **Read:** Performed using a GET request, which retrieves data from a specified resource.
* **Update:** Performed using a PUT request, which is used to update or replace an existing resource.
* **Delete:** Performed using a DELETE request, which is used to remove a specified resource.

*Explore POSTMAN (optional)*

Postman is a powerful platform for API development, testing, and management. It provides a graphical user interface that allows developers to easily send and analyse HTTP requests without writing code. Key features include grouping requests into collections and folders for organization and using variables and a scripting environment for automation and testing.

*Optimization and Efficiency*

API performance optimization is crucial for improving speed, scalability, and reliability. Key principles include:

* **Minimize Latency:** Reduce the time it takes for an API to respond through techniques like caching and batch processing.
* **Scale Efficiently:** Design the API to handle increasing traffic without sacrificing performance.
* **Use Caching Strategically:** Store frequently used data in memory to reduce the need for slower disk or network access.
* **Optimize Resource Utilization:** Use resources like CPU and memory efficiently with techniques such as load balancing and connection pooling.

*Requests lib in Python*

The requests library is a popular third-party package for making HTTP requests in Python. It is not a built-in module and must be installed separately using

pip. It provides a simple API that handles the complexity of the underlying HTTP protocol, allowing developers to focus on their application's logic.

*RBAC (optional)*

**Role-Based Access Control (RBAC)** is a security model that manages access rights by assigning permissions to users based on their role within an organization, rather than individually. This approach simplifies administration, improves security, and is highly scalable. In the RBAC model, users are assigned one or more **roles**, which are collections of **permissions** that define what actions a user can perform on a given **resource**. This is particularly useful in multi-tenant or SaaS products where a user may have a privileged role in one organization but not another.

***Extra***

*Software Development Life Cycle (SDLC)*

The **Software Development Life Cycle (SDLC)** is a structured, step-by-step process used to design, develop, and test high-quality software. The SDLC provides a clear roadmap from a project's initial concept to its maintenance. It typically consists of seven key phases:

1. **Planning:** Defining the project's purpose, scope, and feasibility.
2. **Requirements Analysis:** Gathering and documenting the precise needs of end-users and stakeholders.
3. **Design:** Outlining the software's architecture, including its structure, user interfaces, and database design.
4. **Coding:** The actual development phase where engineers translate the design into functional code.
5. **Testing:** A rigorous quality inspection to uncover bugs and verify that the software meets its requirements.
6. **Deployment:** The process of releasing the software to its end-users, often with a specific strategy to minimize disruption.
7. **Maintenance:** The final phase of ongoing support, bug fixes, and continuous improvements to ensure the software's longevity.

*Agile Basics*

Agile is an incremental and non-linear approach to project management that focuses on flexibility and adaptability. It is a framework for breaking down large projects into smaller, manageable tasks that are completed in short iterations called

**sprints**. The Agile Manifesto is built on four core values:

* **Individuals and interactions** over processes and tools.
* **Working software** over comprehensive documentation.
* **Customer collaboration** over contract negotiation.
* **Responding to change** over following a plan.

Key Agile practices include daily stand-up meetings to keep teams aligned, sprint planning to determine what will be accomplished, and regular retrospectives where teams reflect on their performance.

*Version Control*

A **Version Control System (VCS)** is a software tool that helps developers manage changes to source code over time. It allows a team to track different versions of a codebase, revert to previous states, and collaborate on a single project.

It is important to understand the distinction between **Git** and **GitHub**. Git is the open-source, local version control tool that tracks changes to files. GitHub is a company that provides a popular web-based platform for hosting remote Git repositories, which simplifies collaboration and code sharing.

*Software Architecture*

Architectural principles are a set of guidelines that help create code that is modular, extensible, and maintainable.

* **DRY (Don't Repeat Yourself) Principle:** This principle states that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system". The core idea is to avoid code duplication, which makes the codebase easier to maintain, as changes or bug fixes only need to be applied in one place.
* **SOLID Principles:** A set of five design principles that help create robust and extensible codebases.
  + **Single Responsibility Principle (SRP):** A class or module should have only one reason to change, meaning it should focus on doing one thing well.
  + **Open/Closed Principle (OCP):** Software entities should be "open for extension but closed for modification". This means new features should be added by extending existing code (e.g., via inheritance or polymorphism) rather than by modifying it directly.
  + **Liskov Substitution Principle (LSP):** A subclass must be able to be substituted for its parent class without affecting the system's consistency.