ApexaiQ Pvt. Ltd.

Documentation on Python, APIs, and Coding Standards

Prepared by: Yash Ramesh Fating

Date: 25-Sept-2025

Python :

WHAT IS PYTHON?

=> Python is a simple and easy to understand language which feels like reading simple English. This Pseudo code nature is easy to learn and understandable by beginners

Primarily Data Types in Python :

1. Integers
2. Floating point numbers
3. Strings
4. Booleans
5. None

Following are some common operators in python:

1. Arithmetic operators: +, -, *, / etc.

2. Assignment operators: =, +=, -= etc.

3. Comparison operators: ==, >, >=,

Strings : -

String is a data type in python. String is a sequence of characters enclosed in quotes. We can primarily write a string in these three ways.

```
a ='harry' # Single quoted string

b = "harry" # Double quoted string

c = '''harry''' # Triple quoted string
```

Following are the some of the String Functions :

Function	Description	Example	Output
len()	Returns length of string	len("harry")	5
str.endswith("rry")	Checks if string ends with given substring	"harry".endswith("rry")	true
str.count("r")	Counts occurrences of a character	"harry".count("r")	2
str.capitalize()	Capitalizes first character	"harry".capitalize()	"Harry"
str.find("rr")	Finds index of first occurrence	"harry".find("rr")	2
str.replace("r","l")	Replaces substring with new one	"harry".replace("r","l")	"hally"

#List , Tuple, Dict And Sets

1. List

- **Definition**: Ordered, mutable (changeable), allows duplicates.
- **Syntax**: [ ]

```
fruits = ["apple", "banana", "mango"]

fruits.append("grape")   # Add element

print(fruits)
```

Output : ['apple', 'banana', 'mango', 'grape']

2. Tuple

- **Definition**: Ordered, immutable (cannot change), allows duplicates.
- **Syntax**: ( )

```
numbers = (1, 2, 3, 2)

print(numbers[1])
```

Output : 2

3. Dictionary

- **Definition**: Key–Value pairs, unordered, mutable, keys must be unique.
- **Syntax**: {}

```
student = {"name": "John", "age": 20}

student["age"] = 21   # Update value

print(student)
```

Output : {'name': 'John', 'age': 21}

4. Set

- **Definition**: Unordered, mutable, **no duplicates**.
- **Syntax**: {}

```
unique_nums = {1, 2, 3, 2, 1}
print(unique_nums)  # {1, 2, 3}
```

Conditional Statements :

They let you run code only if certain conditions are true.

Types:

1. if → runs block if condition is true
2. if-else → runs one block if true, another if false
3. if-elif-else → checks multiple conditions

---

1. if Statement

```
x = 10
if x > 5:
    print("x is greater than 5")
# Output: x is greater than 5
```

2. if-else Statement

```
x = 10
if x % 2 == 0:
    print("Even")
else:
    print("Odd")
# Output: Even
```

3. if-elif-else Statement

```
x = 10
if x < 0:
   print("Negative")
elif x == 0:
   print("Zero")
else:
   print("Positive")
# Output: Positive
```

Loops :

Loops let us execute a block of code repeatedly until a condition is met.

🔷 Types of Loops in Python

1. for loop → Iterates over a sequence (list, string, range, etc.).
2. while loop → Repeats as long as a condition is true.
3. Loop control statements → break, continue, pass.

📝 Examples

1. for Loop

```
for i in range(5):
   print(i)
# Output: 0 1 2 3 4
```

2. while Loop

```
x = 1
while x <= 5:
```

```
    print(x)
    x += 1
# Output: 1 2 3 4 5
```

3. break Statement (stop loop early)

```
for i in range(10):
    if i == 5:
        break
    print(i)
# Output: 0 1 2 3 4
```

4. continue Statement (skip current iteration)

```
for i in range(5):
    if i == 2:
        continue
    print(i)
# Output: 0 1 3 4
```

5. pass Statement (placeholder, does nothing)

```
for i in range(3):
    pass   # to be implemented later
```

Functions :

A function is a reusable block of code that performs a specific task.

- Helps avoid code repetition.
- Makes programs cleaner and modular.

🔷 Types of Functions

1. Built-in functions → e.g., len(), print(), sum()

2. User-defined functions → created using def keyword

📝 Examples

1. Simple Function

```
def greet():
    print("Hello, Python!")


greet()
# Output: Hello, Python!
```

2. Function with Parameters

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


print(add(5, 3))
# Output: 8
```

3. Default Arguments

```
def greet(name="User"):
    print("Hello,", name)


greet()        # Output: Hello, User
greet("Alice")  # Output: Hello, Alice
```

4. *args (Multiple Arguments)

```
def total(*nums):
    return sum(nums)


print(total(2, 3, 4))
```

```
# Output: 9
```

5. **kwargs (Keyword Arguments)

```
def info(**data):
  for key, value in data.items():
    print(key, ":", value)

info(name="John", age=25)
# Output:
# name : John
# age : 25
```

Exception Handling :

- An exception is an error that occurs during program execution.
- Exception handling allows us to deal with such errors gracefully without crashing the program.

🔷 Keywords Used

- try → Block of code to test.
- except → Block of code to handle errors.
- else → Runs if no exception occurs.
- finally → Runs always (for cleanup, closing files, etc.).

📝 Examples

1. Basic try-except

```
try:
  x = 10 / 0
```

```
except ZeroDivisionError:
    print("Cannot divide by zero!")
# Output: Cannot divide by zero!
```

2. Multiple except Blocks

```
try:
    num = int("abc")
except ValueError:
    print("Invalid conversion!")
except ZeroDivisionError:
    print("Division error!")
# Output: Invalid conversion!
```

3. Using else

```
try:
    num = int("100")
except ValueError:
    print("Error!")
else:
    print("Conversion successful:", num)
# Output: Conversion successful: 100
```

4. Using finally

```
try:
    f = open("data.txt", "r")
    content = f.read()
except FileNotFoundError:
    print("File not found!")
finally:
```

```
    print("Execution complete (cleanup here).")
```

Decorators :

- A **decorator** is a function that modifies the behavior of another function without changing its code.
- They are commonly used for **logging, authentication, timing, debugging**, etc.

🔷 How Decorators Work

1. A function is passed as an argument to another function.
2. The outer function adds extra functionality.
3. The decorator returns the modified function.

📝 Examples

1. Basic Decorator

```
def my_decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
# Output:
```

```
# Before function call
# Hello!
# After function call
```

---

2. Decorator with Arguments

```
def repeat(func):
    def wrapper(*args, **kwargs):
        print("Calling function twice:")
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper

@repeat
def greet(name):
    print("Hello,", name)

greet("Alice")
# Output:
# Calling function twice:
# Hello, Alice
# Hello, Alice
```

---

3. Using Multiple Decorators

```
def star(func):
    def wrapper():
        print("*****")
```

```
        func()
        print("*****")
    return wrapper

def exclaim(func):
    def wrapper():
        print("!!!")
        func()
        print("!!!")
    return wrapper

@star
@exclaim
def message():
    print("Python Rocks")

message()
# Output:
# *****
# !!!
# Python Rocks
# !!!
# *****
```

Object-Oriented Programming (OOPS)

- **Object-Oriented Programming (OOP)** is a way of structuring programs using **classes** and **objects**.
- It helps in **code reusability, modularity, and organization**.

🔷 Main OOP Concepts

1. **Class** → Blueprint for creating objects.
2. **Object** → Instance of a class.
3. **Inheritance** → Reuse code by deriving classes from others.
4. **Polymorphism** → Same function/method works differently in different contexts.
5. **Encapsulation** → Hide details; expose only what's needed.
6. **Abstraction** → Show essential features, hide implementation.

📝 Examples

1. Class & Object

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self. model = model

    def show(self):
        print(self.brand, self.model)

my_car = Car("Toyota", "Corolla")
my_car.show()
# Output: Toyota Corolla
```

---

2. Inheritance

```
class Animal:
    def sound(self):
        print("Some sound")

class Dog(Animal):   # Inherits from Animal
    def sound(self):
        print("Bark")

d = Dog()
d.sound()
# Output: Bark
```

---

3. Polymorphism

```
class Cat:
    def sound(self): print("Meow")

class Dog:
    def sound(self): print("Bark")

for animal in (Cat(), Dog()):
    animal.sound()
# Output:
# Meow
# Bark
```

---

4. Encapsulation (Private Variable)

```
class Student:
    def __init__(self, name):
        self.__name = name  # private variable

    def get_name(self):
        return self.__name

s = Student("Alice")
print(s.get_name())
# Output: Alice
```

---

5. Abstraction (with ABC)

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, r):
        self.r = r
    def area(self):
        return 3.14 * self.r * self.r
```

```
c = Circle(5)

print(c.area())

# Output: 78.5
```

Comprehensions :

- A **short and elegant way** to create new sequences (lists, sets, dicts) from existing ones.
- Makes code **concise, readable, and faster**.

🔷 Types of Comprehensions

1. **List Comprehension**
2. **Set Comprehension**
3. **Dictionary Comprehension**
4. **Generator Expression**

📝 Examples

1. List Comprehension

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

squares = [x*x for x in nums]

print(squares)

# Output: [1, 4, 9, 16, 25]
```

---

2. Set Comprehension

```
nums = [1, 2, 2, 3, 4]

unique_squares = {x*x for x in nums}

print(unique_squares)

# Output: {16, 1, 4, 9}
```

---

3. Dictionary Comprehension

```
nums = [1, 2, 3, 4]
square_dict = {x: x*x for x in nums}
print(square_dict)
# Output: {1: 1, 2: 4, 3: 9, 4: 16}
```

---

4. Generator Expression

```
nums = [1, 2, 3, 4]
gen = (x*x for x in nums)   # round brackets
print(next(gen))  # 1
print(next(gen))  # 4
```

Iterators & Generators :

Iterator

- An **iterator** is an object that can be iterated (looped) over.
- Implements two methods:
    - __iter__() → returns iterator object
    - __next__() → returns next element (raises StopIteration when finished)

Example:

```
nums = [1, 2, 3]
it = iter(nums)       # create iterator
print(next(it))       # 1
print(next(it))       # 2
print(next(it))       # 3
```

✅ **Generator**

- A **generator** is a function that yields values one at a time using yield keyword.
- More memory-efficient than iterators created manually.
- Automatically implements **iterator protocol**.

Example 1: Basic Generator

```
def my_gen():
    yield 1
    yield 2
    yield 3


gen = my_gen()
print(next(gen))   # 1
print(next(gen))   # 2
print(next(gen))   # 3
```

Virtual Environments & pip :

✅ **Virtual Environment**

- A **self-contained Python environment** to manage project-specific packages.
- Keeps projects **isolated**, avoids package conflicts.
- Create & activate:

```
python -m venv myenv   # create
source myenv/bin/activate  # Linux/Mac
myenv\Scripts\activate     # Windows
```

✅ **pip**

- **pip** is Python's package manager.

- Used to **install, upgrade, and remove packages**.

pip install package_name # install

pip install --upgrade package_name # upgrade

pip uninstall package_name # remove

Standard Libraries :

A collection of built-in modules and packages that come with Python.

Provides ready-to-use functions for common tasks like file handling, math, dates, and more.

No need to install separately**.**

1.Math

2.random

3. datetime

4. os

5. sys

Coding Standards

Naming Conventions :

- Use **clear, descriptive names** for variables, functions, classes, and modules.
- Common conventions:
  - **Variables & Functions** → snake_case → user_name
  - **Classes** → PascalCase → StudentInfo
  - **Constants** → UPPER_CASE → MAX_SCORE

Docstring :

- **Docstrings** are **inline documentation** for modules, classes, or functions.
- Use **triple quotes** """ """ to describe purpose and usage.

Example:

```
def add(a, b):
    """
    Function to add two numbers.
    Parameters: a, b (int/float)
    Returns: sum of a and b
    """
    return a + b
```

Comments :

- Comments explain **why** code does something, not **what** it does.
- Use # for single-line comments.

Example:

```
# This function calculates area of a circle
def area(radius):
    return 3.14 * radius * radius
```

Types of Testing :

- **Unit Testing** → Tests individual functions/modules
- **Integration Testing** → Tests combined modules
- **System Testing** → Tests entire application
- **Acceptance Testing** → Verifies if system meets requirements

---

PEP8 :

- PEP8 is the **Python Enhancement Proposal 8** – official style guide.
- Key points:
    - 4-space indentation (no tabs)

- Max line length: 79 characters
- Use blank lines to separate functions/classes
- Consistent naming conventions

2.6 SOLID & DRY Principles

- **SOLID Principles** → Guidelines for writing **clean, maintainable OOP code**:
  1. **S** → Single Responsibility
  2. **O** → Open/Closed Principle
  3. **L** → Liskov Substitution
  4. **I** → Interface Segregation
  5. **D** → Dependency Inversion
- **DRY Principle → Don't Repeat Yourself**
  - Reuse code via functions/classes instead of duplicating logic.

APIs Topics

API :

API (Application Programming Interface)

An API is a set of rules and protocols that allows different software applications to communicate with each other. It acts as a bridge between two systems, enabling them to exchange data or perform operations without needing to know the internal details of each other.

Key Points:

- Abstraction: Users can interact with software without knowing its internal workings.
- Types:
  - Web APIs: Communicate over the internet using HTTP/HTTPS (e.g., REST, SOAP).
  - Library/Framework APIs: Allow programs to use pre-defined functions in a library.

- Use Cases:
    - Fetching data from servers (e.g., weather data, social media feeds).
    - Integrating third-party services (e.g., payment gateways, maps).
    - Automating tasks between applications.

Types of APIs

APIs can be categorized based on access, use, and architecture:

1. Based on Access:
    - Open/Public API: Available for anyone to use (e.g., Twitter API).
    - Private API: Restricted to internal use within an organization.
    - Partner API: Shared with specific business partners.
2. Based on Architecture:
    - REST API: Uses HTTP requests; lightweight and popular.
    - SOAP API: Uses XML for messaging; more secure and standardized.
    - GraphQL API: Allows clients to request exactly the data they need.
3. Based on Use:
    - Web APIs: Enable communication over the internet.
    - Library/Framework APIs: Functions provided by software libraries.
    - Hardware APIs: Allow software to interact with hardware devices.

Example:

- A payment gateway API (like PayPal API) is a Partner API and a Web API.

HTTP Status Codes :

200 OK – Request succeeded and response contains the requested data.

201 Created – Resource successfully created (commonly used in POST requests).

301 Moved Permanently – URL has been permanently moved to a new location.

302 Found – URL temporarily redirected to another location.

400 Bad Request – Server cannot process the request due to client error.

401 Unauthorized – Authentication is required or has failed.

403 Forbidden – Server understood the request but refuses to authorize it.

404 Not Found – Requested resource does not exist on the server.

500 Internal Server Error – Server encountered an unexpected condition.

503 Service Unavailable – Server is temporarily unable to handle the reques

Response Formats

JSON (JavaScript Object Notation):

- Lightweight, human-readable, widely used.
- Example:
- {
- "name": "John",
- "age": 30
- }

XML (eXtensible Markup Language):

- Structured, supports complex data, machine-readable.
- Example:
- <person>
- <name>John</name>
- <age>30</age>
- </person>

HTML:

- Web page content sent as a response.
- Example: <h1>Welcome</h1>

Plain Text:

- Simple text without formatting.
- Example: Hello, World!

Types of API Authentication :

API authentication ensures that only authorized users or applications can access an API. Common types include:

1. API Key Authentication:
    - A unique key is provided to the client and sent with each request.
    - Simple but less secure; often used for public APIs.
2. Basic Authentication:
    - Uses username and password encoded in base64 and sent with requests.
    - Easy to implement but should be used with HTTPS only.
3. OAuth (Open Authorization):
    - Token-based authentication allowing limited access without sharing credentials.
    - Commonly used by social media APIs (e.g., Google, Facebook).
4. Bearer Token Authentication:
    - Client sends a token in the request header; server validates it.
    - Often used with OAuth 2.0.
5. JWT (JSON Web Token) Authentication:
    - Encodes user info in a token; server validates the token for secure communication.
    - Lightweight and widely used in modern web APIs

Versioning and Security

1. API Versioning:

API versioning is the practice of managing changes in an API without breaking

existing clients. It allows developers to introduce new features or fixes while maintaining compatibility.

Common Versioning Methods:

- URI Versioning: Include version number in the URL, e.g., /api/v1/users.
- Header Versioning: Specify version in HTTP headers, e.g., Accept: application/vnd.example.v1+json.
- Query Parameter Versioning: Use a query parameter, e.g., /api/users?version=1.

2. API Security:
API security ensures that only authorized users can access the API and that data remains protected.

Key Security Practices:

- Authentication & Authorization: Use API keys, OAuth, JWT, etc.
- HTTPS: Encrypt data in transit.
- Rate Limiting & Throttling: Prevent abuse and denial-of-service attacks.
- Input Validation & Sanitization: Protect against injection attacks.
- CORS (Cross-Origin Resource Sharing) Policies: Control which domains can access the API.

CRUD Operations

CRUD operations are the basic **actions performed on data** in databases or via APIs. The acronym **CRUD** stands for:

1. **Create:** Add new data or records.
    - Example: Adding a new user to the database.
    - HTTP Method: **POST**
2. **Read:** Retrieve or view existing data.
    - Example: Fetching user details.
    - HTTP Method: **GET**
3. **Update:** Modify existing data.
    - Example: Changing a user's email address.
    - HTTP Method: **PUT** or **PATCH**

4. **Delete:** Remove data or records.
    - Example: Deleting a user account.
    - HTTP Method: **DELETE**

POSTMAN

Postman is a popular API development and testing tool that allows developers to **send requests, inspect responses, and automate API workflows** without writing code. It simplifies testing and debugging APIs during development.

Key Features:

- **Request Building:** Create and send HTTP requests (GET, POST, PUT, DELETE).
- **Response Inspection:** View response status, headers, and body in various formats (JSON, XML, HTML).
- **Collections:** Organize API requests into collections for reuse and sharing.
- **Environment Variables:** Store reusable values like API keys, tokens, or URLs.
- **Automation & Testing:** Write scripts for automated testing and validation of API responses.
- **Collaboration:** Share collections and environments with team members.

Optimization and Efficiency

API optimization refers to improving the **performance, efficiency, and scalability** of an API to ensure faster responses, lower resource usage, and better user experience.

Key Techniques for API Optimization:

1. **Caching:** Store frequent responses temporarily to reduce server load and response time.
2. **Pagination:** Send data in chunks instead of returning large datasets at once.
3. **Compression:** Compress response data (e.g., using GZIP) to reduce bandwidth usage.
4. **Efficient Queries:** Optimize database queries and avoid unnecessary operations.

5. **Rate Limiting:** Control request frequency to prevent server overload.
6. **Load Balancing:** Distribute API requests across multiple servers for scalability.
7. **Asynchronous Processing:** Handle time-consuming tasks asynchronously to improve response speed.

Requests Library in Python

The **Requests** library is a popular Python library used to **send HTTP requests** easily. It allows developers to interact with web APIs by sending GET, POST, PUT, DELETE requests and handling responses in a simple way.

Key Features:

- **Send HTTP Requests:** Supports GET, POST, PUT, DELETE, PATCH, etc.
- **Handle Responses:** Access response status, headers, and content.
- **Send Data:** Send form data, JSON, or files in requests.
- **Authentication:** Supports Basic Auth, OAuth, and custom headers.
- **Session Management:** Maintain cookies and session state across requests.

Example:

```
import requests


response = requests.get("https://api.example.com/users")
if response.status_code == 200:
  print(response.json())
```

RBAC (Role-Based Access Control)

RBAC is a method of regulating access to computer systems or applications based on the **roles of individual users** within an organization. Instead of assigning permissions to each user, permissions are assigned to roles, and users are assigned to these roles.

Key Features:

- **Roles:** Define a set of permissions (e.g., Admin, Editor, Viewer).

- **Users:** Assigned one or more roles.
- **Permissions:** Define what actions a role can perform (e.g., read, write, delete).
- **Least Privilege:** Users get only the access necessary for their role.

Example:

- An **Admin** can create, edit, and delete data.
- An **Editor** can edit existing data but cannot delete it.
- A **Viewer** can only read data.

SDLC (Software Development Life Cycle):
SDLC is a structured process used to develop software efficiently and with high quality. It includes phases like **Requirement Analysis, Design, Implementation, Testing, Deployment, and Maintenance**.
Key Point: Ensures systematic development and reduces project risks.

Agile Basics:
Agile is a **flexible software development methodology** that emphasizes iterative progress, collaboration, and customer feedback.

- Uses short cycles called **sprints**.
- Promotes **continuous improvement** and adaptability.
  Key Point: Agile delivers functional software quickly while accommodating changes.

Version Control:
Version control is a system that **tracks changes in code or files** over time.

- Examples: **Git, SVN**
- Allows **collaboration, rollback, and branching**.
  Key Point: Helps manage code efficiently in team projects.

Software Architecture:
Software architecture defines the **high-level structure** of a software system, including components, their relationships, and design principles.

- Examples: **Monolithic, Microservices, Client-Server**
  Key Point: Good architecture ensures scalability, maintainability, and performance.