ApexaiQ Internship Documentation

Prepared by: Swami Ganesh Deshpande
Intern - ApexaiQ

1]. Python Syntax, Variables & Data Types

Python Syntax

Python syntax defines how code should be written and structured. Unlike many programming languages that use curly braces {} to define code blocks, Python uses **indentation (spaces or tabs)**. Indentation improves code readability and helps identify logical blocks such as loops, conditionals, or functions.

Example:

```
if True:
   print("Python uses indentation for code blocks")
```

Rules:

- Use **4 spaces** for indentation.
- Python statements end automatically at the end of a line (no need for semicolons).
- Case-sensitive language (Name and name are different).

Variables

Variables are used to store data values. In Python, you don't need to declare variable types explicitly; the interpreter infers the type automatically.

Example:

```
name = "Swami"
age = 20
is_student = True
```

Rules for Naming Variables:

- Must start with a letter or underscore.
- Cannot start with a number.
- Cannot contain special characters like @, $, %.
- Use **snake_case** for naming (e.g., student_name).

Data Types

Python provides several built-in data types:

Type	Example	Description
int	10	Whole numbers
float	3.14	Decimal numbers
str	"Swami"	Text or string values
bool	True / False	Boolean values
list	[1, 2, 3]	Ordered, mutable collection
tuple	(1, 2, 3)	Ordered, immutable collection

dict	{"name": "Swami", "age": 20}
set	{1, 2, 3}

Example:

```
student = {
   "name": "Swami",
   "roll": 72,
   "skills": ["Python", "Java", "HTML"]
}
print(student["skills"][0])
```

2]. Conditional Statements (if-elif-else)

Conditional statements help you control the flow of a program based on certain conditions.

Example:

```
marks = 85

if marks >= 90:
   print("Excellent")
Elif marks >= 75:
   print("Good")
Elif marks >= 50:
   print("Average")
else:
   print("Fail")
```

Explanation:

- if executes the block if the condition is true.
- Elif checks the next condition when the previous one fails.
- else executes when all conditions are false.

3]. Loops (for, while, break, continue)

Loops are used to repeat a block of code multiple times.

For Loop

Used to iterate over sequences like lists, strings, or ranges.

```
for i in range(1, 6):
   print("Number:", i)
```

While Loop

Runs as long as the condition is true.

```
count = 0
while count < 5:
   print("Count:", count)
   count += 1
```

Break and Continue

- **break:** exits the loop completely.
- **continue:** skips the current iteration.

```
for i in range(1, 10):
   if i == 5:
      break
   if i == 2:
      continue
```

```
    print(i)
```

4]. Functions

Functions are reusable blocks of code that perform a specific task.

Basic Function

```
def greet(name):
    print(f"Hello, {name}!")
greet("Swami")
```

Return Statement

Functions can return a value to the caller.

```
def add(a, b):
    return a + b
print(add(10, 20))
```

Default Parameters

```
def welcome(name="User"):
    print("Welcome,", name)
welcome()
welcome("Swami")
```

Variable Arguments (*args, kwargs)

```
def show_details(*args, **kwargs):
    print(args)
    print(kwargs)
show_details(1, 2, 3, name="Swami", age=20)
```

5].Exception Handling

Exception handling helps manage runtime errors to avoid crashing the program.

Example:

```
try:
    num = int(input("Enter number: "))
    print(10 / num)
except ZeroDivisionError:
    print("Error: Division by zero is not allowed!")
except ValueError:
    print("Error: Please enter a valid number.")
finally:
    print("Execution completed.")
```

Keywords:

- try → block where you test code.
- except → handles errors.
- finally → always executes (used for cleanup).

6]. Decorators

Decorators modify the behavior of a function without changing its code.

They are often used for **logging, authentication, or performance measurement.**

Example:

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

@decorator
def hello():
    print("Hello World!")

hello()
```
Output:

```
Before function
Hello World!
After function
```

7]. OOPS (Object-Oriented Programming)

OOP is a programming style based on objects and classes.

Class and Object Example

```
class Student:
    def __init__(self, name, roll):
        self.name = name
        self.roll = roll

    def display(self):
        print(f"Name: {self.name}, Roll No: {self.roll}")

s1 = Student("Swami", 72)
s1.display()
```
OOP Concepts

- **Encapsulation:** Binding data and methods together.
- **Inheritance:** Deriving new classes from existing ones.
- **Polymorphism:** Same function name used for different purposes.
- **Abstraction:** Hiding unnecessary details.

Example of Inheritance:

```
class Person:
    def speak(self):
        print("I am a person.")

class Student(Person):
    def study(self):
        print("I am studying.")

obj = Student()
obj.speak()
obj.study()
```

8]. List & Dictionary Comprehension

A shorter way to create lists or dictionaries.

```
squares = [x**2 for x in range(5)]
```

```
print(squares)

ages = {"Swami": 20, "Ganesh": 22}
adult = {k: v for k, v in ages.items() if v > 20}
print(adult)
```

9].Iterators & Generators

Iterators

An iterator allows you to traverse through elements.

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

Generators

Used to generate values one by one using yield.

```
def gen():
   for i in range(3):
       yield i
for x in gen():
   print(x)
```

10].Virtual Environments & pip

A **virtual environment** is an isolated Python setup for projects.
It avoids version conflicts between libraries.

Commands:

```
python -m venv env
env\Scripts\activate
source env/bin/activate
pip install requests
```

Deactivate using:

```
deactivate
```

11]. Standard Libraries

Python provides many built-in modules to simplify coding.

Library	Description
math	Mathematical functions
datetime	Date and time manipulation
os	Operating system functions
json	JSON data parsing
random	Generate random values

Example:

```
import datetime
print(datetime.datetime.now())
```

1]. Naming Conventions

A **naming convention** defines rules for naming variables, functions, classes, and constants to make the code **readable and maintainable**.

Good naming helps other developers understand the purpose of a variable or function immediately.

Best Practices:

Category	**Convention**	**Example**
Variable	lowercase_with_underscores	student_name, total_marks
Function	lowercase_with_underscores	calculate_average(), get_user_input()
Class	PascalCase	StudentDetails, UserAccount
Constant	ALL_CAPS_WITH_UNDERSCORES	MAX_SIZE = 100, PI = 3.14
Module/File	lowercase	math_utils.py

Example:

```
class StudentInfo:
    MAX_MARKS = 100

    def __init__(self, name, marks):
        self.name = name
        self.marks = marks

    def display_info(self):
        print(f"Student: {self.name}, Marks: {self.marks}")
```

Why Important?

→ Improves readability, teamwork, and reduces confusion between variables and functions.

2]. Docstrings

A **Docstring (Documentation String)** explains what a function, class, or module does.
It is written inside triple quotes (""" """) and is used by tools like IDEs and Sphinx to generate documentation automatically.

Function Docstring Example:

```
def add(a, b):
    """
    Adds two numbers and returns the result.

    Parameters:
    a (int): First number
    b (int): Second number

    Returns:
    int: Sum of a and b
    """
    return a + b
```

Class Docstring Example:

```
class Calculator:
    """A simple calculator class for basic math operations."""

    def multiply(self, a, b):
        """Return product of two numbers."""
        return a * b
```

Why Important?

- Helps new developers understand code purpose.
- Used by help() function in Python.
- Promotes clean, self-explanatory code.

3]. Comments
Comments are lines ignored by the Python interpreter, used to explain code logic.
Types of Comments:

1. **Single-line Comment:**
2. # This function adds two numbers
3. def add(a, b):
4.   return a + b
5. **Multi-line Comment:**
6. """
7. This block describes
8. the program purpose.

4]. Types of Testing
Testing ensures your code works correctly and avoids bugs before release.
Here are the **main testing types used in software development**:

Test Type	**Description**	**Example**
Unit Testing	Tests individual functions or modules	Test add() function
Integration Testing	Tests how multiple modules work together	Check login + dashboard connection
System Testing	Tests the complete system as a whole	Entire web app testing
Acceptance Testing	Ensures product meets client requirements	Final project demo test
Regression Testing	Checks that new updates don't break old features	Test app after API version update

Example (Unit Test using unittest module):
```
import unittest

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

class TestMath(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)

if __name__ == '__main__':
    unittest.main()
```

5]. PEP8 Guidelines
PEP8 is the **official style guide for Python** created by the Python community.
It defines best practices for code layout and formatting.
Main PEP8 Rules:

1. Use **4 spaces per indentation level**.
2. Keep lines **under 79 characters** long.
3. Leave **blank lines** between functions and classes.
4. Use **meaningful variable names**.
5. Keep imports at the **top of the file**.
6. Use **spaces around operators**:

7. total = price + tax
8. Avoid unnecessary spaces inside parentheses:
   print(x) → print(x)

Example (Bad vs Good):

Bad Code:

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

Good Code (PEP8 Compliant):

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

1]. What is an API?

An **API (Application Programming Interface)** is a set of **rules and protocols** that allows different software applications to communicate with each other.

Think of it as a **bridge** between two systems — for example, when you use a food delivery app, the app sends a request to the restaurant's server using APIs to fetch menu data and place orders.

Example:

When you visit a weather app, it uses an API like:

https://api.openweathermap.org/data/2.5/weather?q=London&appid=your_api_key

The app sends a request to the server, and the server responds with data (like temperature, humidity, etc.) in **JSON** format.

Python Example using requests:

```
import requests

response = requests. Get("https://api.github.com")
print("Status Code:", response.status_code)
print("Response Body:", response. Json())
```

This code sends a GET request to GitHub's API and prints the server's response.

2]. Types of APIs

APIs can be categorized based on **access level** and **functionality**.

Based on Access:

Type	Description	Example
Open/Public API	Accessible to everyone; requires minimal authentication.	Google Maps API, Weather API
Partner API	Shared between specific business partners.	Paytm → Bank API
Private API	Used internally within a company.	Internal employee management system
Composite API	Combines multiple APIs into one call.	E-commerce checkout (fetch user, product, and payment info together)

Based on Functionality:

- **Web APIs** – Accessed via HTTP/HTTPS over the internet.
- **Library APIs** – Functions provided by software libraries (e.g., Python math library).
- **OS APIs** – For interacting with operating system components (e.g., Windows API).

3]. HTTP Status Codes

HTTP status codes are **responses from the server** indicating whether a request was successful or failed.

Code	Category	Meaning
200	Success	Request successful
201	Created	Resource successfully created
400	Client Error	Bad request
401	Unauthorized	Authentication failed
403	Forbidden	Access not allowed
404	Not Found	Resource not found
500	Server Error	Problem on the server side

Example:

```
response = requests. Get("https://api.github.com/users/swamiD18")
if response.status_code == 200:
   print("Request successful")
else:
   print("Error:", response.status_code)
```

4]. Response Formats

APIs return data in structured formats that are easy to parse.

Format	Description	Example
JSON (JavaScript Object Notation)	Most common, lightweight	{"name": "Swami", "age": 20}
XML (Extensible Markup Language)	Used in older APIs	<user><name>Swami</name></user>
HTML/Text	Used in web scraping or simple responses	<h1>Welcome</h1>

Example (Reading JSON):

```
import requests

r = requests. Get("https://api.github.com/users/swamiD18")
data = r.json()
print(data["login"])
```

5]. Types of API Authentication

Authentication ensures that only authorized users can access the API.

Type	Description	Example
API Key	Simple token passed in URL or header	?apikey=12345
Basic Auth	Uses username and password encoded in Base64	Common in testing APIs
OAuth 2.0	Secure industry standard for third-party access (e.g., Google, Facebook login)	
JWT (JSON Web Token)	Token-based authentication used in web apps	
Bearer Token	Token passed in headers for verification	

Example:

```
headers = {"Authorization": "Bearer your_token_here"}
r = requests. Get("https://api.example.com/user", headers=headers)
print(r.status_code)
```

6].Versioning and Security
API **versioning** ensures backward compatibility when updates are made.
Versioning Example:
https://api.example.com/v1/users
https://api.example.com/v2/users
Here, v1 and v2 are different API versions.
Security Measures:
- Always use **HTTPS** instead of HTTP.
- Validate all input data to prevent **SQL injection** or **XSS**.
- Implement **rate limiting** to prevent abuse.
- Use **tokens** or **API keys** for authentication

7]. CRUD Operations
CRUD stands for **Create, Read, Update, Delete** — the four basic operations used in APIs.

Operation	HTTP Method	Description	Example Endpoint
Create	POST	Add new data	/users
Read	GET	Retrieve data	/users/1
Update	PUT / PATCH	Modify existing data	/users/1
Delete	DELETE	Remove data	/users/1

Example in Python (using requests):
```
import requests

# Create (POST)
data = {"name": "Swami", "age": 20}
r = requests. Post("https://reqres.in/api/users", Json=data)
print("Create:", r.json())

# Read (GET)
r = requests. Get("https://reqres.in/api/users/2")
print("Read:", r.json())

# Update (PUT)
update = {"name": "Swami D", "age": 21}
r = requests. Put("https://reqres.in/api/users/2", json=update)
print("Update:", r.json())

# Delete
r = requests. Delete("https://reqres.in/api/users/2")
print("Delete:", r.status_code)
```

8]. POSTMAN
Postman is a graphical tool used to test APIs without writing code.
It allows you to:
- Send requests (GET, POST, PUT, DELETE)
- Add authentication headers
- View JSON responses
- Save collections for API documentation

Example Use Case:

- Send a POST request to create a new user.
- Send a GET request to verify if the user was created.
- View response in JSON format.

9]. Optimization and Efficiency

API optimization improves **speed**, **security**, and **scalability**.

Best Practices:

1. **Use Pagination** – Limit results (e.g., ?page=2&limit=50)
2. **Implement Caching** – Store frequent results in memory.
3. **Reduce Payload** – Avoid unnecessary fields in responses.
4. **Use Compression (gzip)** – Reduce data transfer size.
5. **Batch Requests** – Combine multiple small requests into one.
6. **Database Optimization** – Use indexing and efficient queries.
7. **HTTP Headers** – Use proper cache-control and expiry headers.

Example of Pagination:

https://api.example.com/users?page=2&limit=10

10]. Requests Library in Python

The requests library is one of the most powerful tools for working with APIs in Python.
It supports all HTTP methods: GET, POST, PUT, DELETE, etc.

GET Example:

```
import requests

response = requests. Get("https://api.github.com/users/swamiD18")
print(response.json())
```

POST Example:

```
data = {"name": "Swami", "job": "Developer"}
response = requests. Post("https://reqres.in/Api/users", json=data)
print(response. Json())
```

PUT Example:

```
update = {"job": "Full Stack Developer"}
response = requests. Put("https://reqres.in/Api/users/2", json=update)
print(response. Json())
```

DELETE Example:

```
response = requests. Delete("https://reqres.in/Api/users/2")
print(response.status_code)
```

11]. RBAC (Role-Based Access Control)

RBAC controls access to system features based on the **user's role**.
It is commonly used in APIs for **authorization** (deciding what users can do).

Example Roles:

Role	Permission
Admin	Create, Read, Update, Delete
Editor	Create, Read, Update
Viewer	Read only

Concept Example:

```
def access dashboard(role):
    if role == "Admin":
        print("Access to all features")
    Elif role == "Editor":
        print("Access to edit content only")
    else:
        print("Read-only access")

access dashboard("Editor")
```

Conclusion
In conclusion, this research provided a complete understanding of Python fundamentals and API development. It covered Python's syntax, datatypes, control statements, functions, exception handling, OOP, and coding standards like PEP8 and SOLID principles. We also explored API concepts, including types of APIs, HTTP methods, authentication, CRUD operations, and security. Overall, this study highlights how Python's simplicity and APIs' connectivity together enable developers to build efficient, secure, and scalable applications for real-world use.