Python CheatSheet

TABLE OF CONTENTS

1. Introduction to Python
- What is Python?
- History & Features
- Applications of Python
- Installing Python
- Running Python Code (IDLE, VS Code, Terminal)

2. Basic Syntax
- Comments
- Variables
- Keywords
- Indentation
- Input/Output (input(), print())

3. Data Types
- Numbers (int, float, complex)
- Strings
- Boolean
- None Type
- Type Conversion
- type() and isinstance()

4. Operators
- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Identity & Membership Operators
- Operator Precedence

5. Control Flow
- if, elif, else
- Nested Conditions
- Ternary Operator

6. Loops
- for Loops
- while Loops
- Loop Control (break, continue, pass)
- range() Function
- List Comprehension

TABLE OF CONTENTS

7. Functions
- Defining & Calling Functions
- Parameters & Arguments
- Default, Keyword & Arbitrary Arguments
- Return Statement
- Lambda Functions
- map(), filter(), reduce()

8. Data Structures
- Lists
  - Creation, Indexing, Slicing
  - Common Methods
- Tuples
  - Immutable Sequences
- Sets
  - Unique Values, Set Operations
- Dictionaries
  - Key-Value Pairs, Methods
- Strings (Advanced)
  - String Formatting (f-strings, format(), %)
  - String Methods

9. Modules and Packages
- Importing Modules
- import vs from
- Creating Custom Modules
- Using pip to Install Packages
- Popular Libraries Overview (NumPy, pandas, requests, etc.)

10. File Handling
- Reading and Writing Files
- Working with File Paths
- with Statement
- CSV and JSON Handling

11. Error Handling
- try, except, finally
- raise Statement
- Common Exceptions

TABLE OF CONTENTS

12. Object-Oriented Programming (OOP)
- Classes and Objects
- __init__ Constructor
- self Keyword
- Methods
- Inheritance
- Encapsulation & Abstraction
- super() Function
- @staticmethod & @classmethod

13. Advanced Topics
- Iterators & Generators
- Decorators
- Recursion
- *args and **kwargs
- Comprehensions (List, Dict, Set)
- Enumerate, Zip

14. Date and Time
- datetime Module
- Formatting Dates
- Timestamp Conversions

15. Math and Statistics
- math Module
- random Module
- statistics Module

16. Regular Expressions
- re Module Basics
- Pattern Matching
- Common Regex Patterns

17. Working with APIs
- requests Library Basics
- GET and POST Requests
- Handling JSON Data

TABLE OF CONTENTS

18. Pythonic Conventions
- PEP8 Style Guide
- Docstrings
- Code Optimization Tips

19. Virtual Environments & Dependency Management
- venv Module
- requirements.txt
- pip freeze

20. Popular Libraries (Overview)
- Data Science: NumPy, pandas, matplotlib, seaborn
- Web Development: Flask, Django
- Automation: selenium, pyautogui, schedule
- Machine Learning: scikit-learn, TensorFlow, PyTorch

21. Debugging & Testing
- Using pdb
- Writing Unit Tests with unittest or pytest

22. Mini Projects
- Calculator
- To-Do List
- Web Scraper
- Weather App using API
- Alarm Clock

23. Interview Questions (Bonus)
- Frequently Asked Python Questions
- Code Challenges

1. INTRODUCTION TO PYTHON

1.1 What is Python?

- Python is a high-level, interpreted, and general-purpose programming language known for its simple syntax and readability.
- ✅ Great for beginners and professionals alike.

1.2 History & Features

- *Created by:* Guido van Rossum
- *Released:* 1991
- **Key Features:**
  - Easy to learn
  - Open-source and free
  - Dynamically typed
  - Huge standard library
  - Portable across platforms
  - Supports OOP and functional programming

1.3 Applications of Python

Python is used in:

- ✅ Web Development (e.g., Django, Flask)
- ✅ Data Science & Machine Learning (e.g., Pandas, NumPy, Scikit-learn)
- ✅ Automation/Scripting
- ✅ Game Development
- ✅ Cybersecurity
- ✅ IoT
- ✅ Desktop & Mobile Apps

1.4 Installing Python

- Go to python.org/downloads
- Download and install the latest version
- Add Python to system PATH during installation

1.5 Running Python Code

- You can run Python code in several ways:

➤ *IDLE (comes with Python)*

```
print("Hello from Coders_Section!")
```

1. INTRODUCTION TO PYTHON

➤ *Terminal / Command Prompt*

```
python script.py
```

➤ *VS Code (or any code editor)*

- Save file as example.py
- Run in terminal:

```
python example.py
```

➤ *Interactive Shell (REPL)*

- Just type python in terminal and start typing Python code directly.

```
>>> 2 + 3
5
```

2. BASIC SYNTAX

2.1 Comments

- Used to explain code; ignored during execution.

```
# This is a single-line comment

"""
This is a
multi-line comment
"""
```

2.2 Variables

- Containers for storing data; no need to declare data types explicitly.

```
name = "Ravi"
age = 20
pi = 3.14
```

2.3 Keywords

- Reserved words in Python that cannot be used as variable names.
- Examples: if, else, for, while, def, class, return, import

```
# Invalid: def = 5 ❌
# Valid:
count = 10
```

- To view all keywords:

```
import keyword
print(keyword.kwlist)
```

2. BASIC SYNTAX

2.4 Indentation

- Python uses indentation (spaces/tabs) to define blocks of code (no {} like other languages).

```
if age > 18:
    print("Adult")  # Indented block
else:
    print("Minor")
```

- ⚠️ Incorrect indentation will raise an IndentationError.

2.5 Input/Output

- ➤ input() – Takes user input (as a string):

```
name = input("Enter your name: ")
print("Hello", name)
```

- ➤ print() – Displays output:

```
print("Python is fun!")
print("Age:", age)
```

- You can also format output:

```
print(f"My name is {name} and I am {age} years old.")
```

3. DATA TYPES

3.1 Numbers (int, float, complex)

Python can work with numbers just like a calculator.

- int is a whole number → 5, 100
- float is a number with a dot → 3.14, 2.0
- complex has a little math magic → 2 + 3j (used in advanced math)

```
a = 5          # int
b = 3.14       # float
c = 2 + 3j     # complex
```

3.2 Strings

- Strings are just text! You put it inside quotes.

```
name = "Rudra"
print(name)
```

3.3 Boolean

- This is a special type that only says True or False — like yes or no!

```
is_sunny = True
is_raining = False
```

3.4 None Type

- “None” means nothing. It’s like an empty box.

```
x = None
print(x)  # It prints: None
```

3. DATA TYPES

3.5 Type Conversion

- You can change one type to another.

```
age = "10"       # This is a string
real_age = int(age)  # Now it's a number!
```

3.6 type() and isinstance()

- These help us check what kind of thing something is.

```
x = 5
print(type(x))          # Says: int
print(isinstance(x, int))  # Says: True
```

4. OPERATORS

4.1 Arithmetic Operators

- These are math symbols:
- + (add), - (subtract), * (multiply), / (divide), // (divide and round down), % (remainder), ** (power)

```
print(5 + 2)   # 7
print(5 ** 2)  # 25
```

4.2 Comparison Operators

- Used to compare numbers:
- == (equal), != (not equal), > (greater), < (less), >=, <=

```
print(10 > 5)   # True
print(3 == 3)   # True
```

4.3 Logical Operators

- Used when you have more than one condition.
- and → both must be true
- or → one can be true
- not → opposite

```
print(True and False)  # False
print(not True)        # False
```

4.4 Assignment Operators

- Used to store values in a variable.

```
x = 5
x += 2   # same as x = x + 2 → x is now 7
```

4. OPERATORS

4.5 Bitwise Operators

- These work with 0s and 1s (like computer brain math) – not needed early on.

4.6 Identity & Membership Operators

- “is” checks if two things are exactly the same
- “in” checks if something is inside something

```
a = [1, 2, 3]
print(2 in a)   # True
```

4.7 Operator Precedence

- Some operations happen first (like multiplication before addition).

```
result = 5 + 2 * 3  # 2*3 first = 6 → 5+6 = 11
```

5. CONTROL FLOW

5.1 if, elif, else

- It helps your code make decisions.

```
age = 10
if age ≥ 18:
    print("Adult")
elif age == 10:
    print("You're 10!")
else:
    print("Child")
```

5.2 Nested Conditions

- An if inside another if.

```
score = 85
if score > 50:
    if score < 90:
        print("Good score!")
```

5.3 Ternary Operator

- A short way to write if else.

```
age = 17
status = "Adult" if age ≥ 18 else "Child"
print(status)
```

6. LOOPS

6.1 for Loops

- When you want to do something again and again, use a loop!

```
for i in range(5):
    print(i)
# Prints 0 to 4
```

6.2 while Loops

- Runs as long as something is True.

```
x = 1
while x < 4:
    print(x)
    x += 1
```

6.3 Loop Control

- break: Stop the loop
- continue: Skip and move to the next loop
- pass: Do nothing (just a placeholder)

```
for i in range(5):
    if i == 3:
        break  # Stops at 3
    print(i)
```

6.4 range() Function

- It gives a list of numbers to loop through.

```
range(5)  # 0, 1, 2, 3, 4
range(2, 6)  # 2, 3, 4, 5
```

6. LOOPS

6.5 List Comprehension

- A short and fast way to create lists.

```
squares = [x*x for x in range(5)]
print(squares)  # [0, 1, 4, 9, 16]
```

7. FUNCTIONS

7.1 Defining & Calling Functions

- Functions help you reuse code.

```
def say_hello():
    print("Hello!")
say_hello()
```

7.2 Parameters & Arguments

- Give inputs to a function.

```
def greet(name):
    print("Hello", name)
greet("Rudra")
```

7.3 Default, Keyword & Arbitrary Arguments

- Default → value used if nothing is given
- Keyword → specify name
- Arbitrary → many arguments using * or **

```
def hello(name="Friend"):
    print("Hi", name)
hello()
hello("Rudra")
```

7.4 Return Statement

- It gives back a value.

```
def add(a, b):
    return a + b
print(add(2, 3))  # 5
```

7. FUNCTIONS

7.5 Lambda Functions

- Tiny functions written in one line.

```
square = lambda x: x * x
print(square(5))  # 25
```

7.6 map(), filter(), reduce()

- Used with functions to work on lists:

```
nums = [1, 2, 3]
doubled = list(map(lambda x: x*2, nums))
print(doubled)  # [2, 4, 6]
```

8. DATA STRUCTURES

8.1 Lists

- Store many items. You can change them.

```
fruits = ["apple", "banana"]
print(fruits[0])  # "apple"
fruits.append("mango")
```

8.2 Tuples

- Like lists, but you can't change them.

```
colors = ("red", "blue")
```

8.3 Sets

- No duplicates allowed!

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

- You can do math with sets like union and intersection.

8.4 Dictionaries

- Think of it like a mini-database — it stores key-value pairs.

```
student = {"name": "Rudra", "age": 19}
print(student["name"])  # Rudra
```

8. DATA STRUCTURES

8.5 Strings (Advanced)

f-strings, format(), %

- Different ways to put variables inside strings:

```
name = "Rudra"
print(f"Hi {name}!")
print("Hi {}".format(name))
print("Hi %s" % name)
```

String Methods

- Useful tools like .upper(), .lower(), .replace()

```
text = "hello"
print(text.upper())  # HELLO
```

9. MODULES AND PACKAGES

9.1 Importing Modules

- Python has ready-made tools called modules.
- You can use them with the import keyword:

```
import math
print(math.sqrt(25))  # 5.0
```

9.2 import vs from

- import math means you use math.sqrt()
- from math import sqrt means just use sqrt() directly

```
from math import sqrt
print(sqrt(16))  # 4.0
```

9.3 Creating Custom Modules

- You can create your own module (a .py file with some functions) and use it in another file.

greetings.py:

```
def say_hello():
    print("Hello!")
```

main.py:

```
import greetings
greetings.say_hello()
```

9. MODULES AND PACKAGES

9.4 Using pip to Install Packages

- pip is like a shopping cart for code tools. You can install stuff like this:

```
pip install numpy
```

9.5 Popular Libraries Overview

- NumPy: Math with big numbers & arrays
- pandas: Handling data like Excel
- requests: Talk to websites & APIs
- matplotlib: Make graphs & charts
- tkinter: Build simple apps with buttons & windows

10. FILE HANDLING

10.1 Reading and Writing Files

- Open files to read or write:

```
# Write to a file
with open("file.txt", "w") as f:
    f.write("Hello!")

# Read the file
with open("file.txt", "r") as f:
    content = f.read()
    print(content)
```

10.2 Working with File Paths

- You can use folders and paths:

```
file = open("my_folder/data.txt")
```

- Or for better safety:

```
import os
path = os.path.join("my_folder", "data.txt")
```

10.3 with Statement

- with is used so you don't forget to close the file.

```
with open("file.txt", "r") as f:
    print(f.read())
```

10. FILE HANDLING

10.4 CSV and JSON Handling

- CSV (Comma Separated Values)
- Used for rows & columns (like Excel):

```python
import csv
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

- JSON (JavaScript Object Notation)
- Used for storing data (like dictionaries):

```python
import json
person = {"name": "Rudra", "age": 19}
json_data = json.dumps(person)
print(json_data)
```

11. ERROR HANDLING

- Sometimes, your code might break. That's called an error or exception. You can catch those errors using special tools.

11.1 try, except, finally

```
try:
    number = int("hello")  # this will crash
except ValueError:
    print("Oops! That's not a number.")
finally:
    print("This will always run.")
```

- try: Test some code
- except: What to do if there's a problem
- finally: Always runs at the end (even if there's an error)

11.2 raise Statement

- You can make your own error with raise.

```
age = -5
if age < 0:
    raise ValueError("Age cannot be negative!")
```

11.3 Common Exceptions

- ValueError: Wrong type of value
- TypeError: Wrong type of data
- ZeroDivisionError: Dividing by 0
- FileNotFoundError: Missing file
- IndexError: Wrong list index
- KeyError: Wrong dictionary key

12. OBJECT-ORIENTED PROGRAMMING (OOP)

- OOP lets us bundle data and functions together. It's like building with LEGO blocks!

12.1 Classes and Objects

- A class is a blueprint. An object is something built from that blueprint.

```
class Dog:
    def bark(self):
        print("Woof!")

my_dog = Dog()
my_dog.bark()
```

12.2 __init__ Constructor

- Runs automatically when an object is created.

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

dog1 = Dog("Bruno")
print(dog1.name)  # Bruno
```

12.3 self Keyword

- self refers to the current object itself.

12.4 Methods

- Functions inside a class are called methods.

```
class Calculator:
    def add(self, a, b):
        return a + b
```

12. OBJECT-ORIENTED PROGRAMMING (OOP)

12.5 Inheritance

- Child classes can use and improve parent class stuff.

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

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

kitty = Cat()
kitty.sound()  # Meow
```

12.6 Encapsulation & Abstraction

- Encapsulation: Hiding the code inside a class
- Abstraction: Showing only what's needed

12.7 super() Function

- Lets you call the parent class method.

```
class Animal:
    def __init__(self):
        print("Animal created")

class Dog(Animal):
    def __init__(self):
        super().__init__()  # calls Animal's __init__
        print("Dog created")
```

12. OBJECT-ORIENTED PROGRAMMING (OOP)

12.8 @staticmethod & @classmethod

- @staticmethod: Doesn't use self, like a normal function inside class
- @classmethod: Uses cls (the class itself)

```python
class MyClass:
    @staticmethod
    def say_hello():
        print("Hi!")

    @classmethod
    def show_class(cls):
        print(cls)

MyClass.say_hello()
MyClass.show_class()
```

13. ADVANCED TOPICS

13.1 Iterators & Generators

- Iterator: Goes through items one by one.

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

- Generator: Like a function that remembers where it left off.

```
def count_up():
    yield 1
    yield 2
gen = count_up()
print(next(gen))  # 1
```

13.2 Decorators

- A decorator is like adding magic to a function — it changes how it works.

```
def greet(func):
    def wrapper():
        print("Hello!")
        func()
    return wrapper

@greet
def say_name():
    print("I'm Python")

say_name()
# Hello!
# I'm Python
```

13. ADVANCED TOPICS

13.3 Recursion

- When a function calls itself.

```
def count(n):
    if n == 0:
        return
    print(n)
    count(n - 1)

count(3)
# 3
# 2
# 1
```

13.4 *args and **kwargs

- *args = many values
- **kwargs = many key=value pairs

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

print(add(1, 2, 3))  # 6

def show_info(**info):
    print(info)

show_info(name="Ravi", age=20)
# {'name': 'Ravi', 'age': 20}
```

13. ADVANCED TOPICS

13.5 Comprehensions (List, Dict, Set)

- Short way to make lists, sets, or dictionaries.

```
squares = [x**2 for x in range(5)]  # List
print(squares)  # [0, 1, 4, 9, 16]

unique = {x for x in [1, 1, 2, 3]}  # Set
print(unique)  # {1, 2, 3}

names = {i: f"User{i}" for i in range(3)}  # Dict
print(names)  # {0: 'User0', 1: 'User1', 2: 'User2'}
```

13.6 enumerate() and zip()

- enumerate() gives index and value.
- zip() joins two lists.

```
for i, val in enumerate(["a", "b"]):
    print(i, val)

names = ["A", "B"]
scores = [90, 80]
for n, s in zip(names, scores):
    print(n, s)
```

14. DATE AND TIME

14.1 datetime Module

- The datetime module helps us work with dates and times.
- We can get the current date/time or create specific dates.

```
from datetime import datetime

# Get current date and time
now = datetime.now()
print(now)  # Example: 2025-05-02 12:34:56.789012

# Create a specific date
birthday = datetime(2005, 10, 5)
print(birthday)  # 2005-10-05 00:00:00
```

14.2 Formatting Dates

- We can format the date to look the way we want using .strftime().

```
# Format current date
print(now.strftime("%d/%m/%Y"))  # 02/05/2025
print(now.strftime("%B %d, %Y"))  # May 02, 2025
print(now.strftime("%I:%M %p"))   # 12:34 PM
```

Format codes example:

- **%d** → Day
- **%m** → Month (number)
- **%B** → Month name
- **%Y** → Year
- **%I:%M %p** → Hour:Minute AM/PM

14. DATE AND TIME

14.3 Timestamp Conversions

- A timestamp is a number that shows date/time in seconds since 1970.

```
# Convert to timestamp
print(now.timestamp())  # 1746164096.789012

# Convert from timestamp
ts = 1746164096
dt = datetime.fromtimestamp(ts)
print(dt)  # 2025-05-02 12:34:56
```

15. MATH AND STATISTICS

15.1 math Module

- The math module gives us access to math functions.

```
import math

print(math.sqrt(16))      # 4.0
print(math.factorial(5))  # 120
print(math.pow(2, 3))     # 8.0
print(math.pi)            # 3.141592 ...
```

- Useful for scientific calculations, square roots, trigonometry, etc.

15.2 random Module

- The random module helps with random values — great for games or data shuffling.

```
import random

print(random.randint(1, 10))      # Random number between 1-10
print(random.choice(['A', 'B']))  # Random choice from list
print(random.random())            # Random float between 0 and 1
```

- Great for games, simulations, and practice apps.

15.3 statistics Module

- The statistics module helps us with data analysis: mean, median, mode, etc.

```
import statistics

data = [10, 20, 30, 40]

print(statistics.mean(data))   # 25.0
print(statistics.median(data)) # 25.0
print(statistics.stdev(data))  # 12.9 ...
```

- Useful when working with data science or analyzing user behavior.

16. REGULAR EXPRESSIONS

- Regular Expressions (called RegEx) help you find patterns in text — like phone numbers, emails, or passwords.

16.1 re Module Basics

- To use RegEx in Python, we use the built-in re module:

```
import re
```

16.2 Pattern Matching

- Let's say you want to check if a word is inside a sentence:

```
import re

text = "My name is Rudra"
match = re.search("Rudra", text)

if match:
    print("Found it!")
else:
    print("Not found.")
```

- You can also find all matches:

```
re.findall("a", "banana")  # ['a', 'a', 'a']
```

16. REGULAR EXPRESSIONS

16.3 Common Regex Patterns

Pattern	What it matches
\d	Any digit (0–9)
\w	Any letter/number/underscore
.	Any character except newline
^	Start of string
$	End of string
*	0 or more repeats
+	1 or more repeats
{n}	Exactly n repeats

- Example: Match email pattern

```
pattern = r"\w+@\w+\.\w+"
re.findall(pattern, "Email me at test@example.com")
# ['test@example.com']
```

17. WORKING WITH APIS

- APIs help your Python code talk to websites or apps to get or send data.

17.1 requests Library Basics

- To use APIs, we need the requests library. Install it with:

```
pip install requests
```

- Then:

```
import requests
```

17.2 GET and POST Requests

- GET → To receive data
- POST → To send data

```
# GET Request
response = requests.get("https://jsonplaceholder.typicode.com/posts")
print(response.json())  # Shows data in JSON

# POST Request
data = {"title": "Hello", "body": "World"}
response = requests.post("https://jsonplaceholder.typicode.com/posts",
json=data)
print(response.status_code)  # 201 = Created
```

17.3 Handling JSON Data

- Most APIs use JSON. Python makes it easy:

```
data = response.json()  # Turns JSON to Python dictionary
print(data[0]["title"])
```

18. PYTHONIC CONVENTIONS

- Being "Pythonic" means writing code the smart and clean way.

18.1 PEP8 Style Guide

- PEP8 is the official guide for writing neat Python code.

Some basic tips:

```python
# Good
def say_hello(name):
    print("Hi", name)

# Bad
def sayhello(name):print("Hi",name)
```

18.2 Docstrings

- Use triple quotes (""" """") to describe what a function/class does.

```python
def greet(name):
    """This function greets the user."""
    print(f"Hello, {name}!")
```

18.3 Code Optimization Tips

- Use list comprehensions: [x*x for x in range(5)]
- Avoid unnecessary loops
- Use built-in functions like sum(), max(), sorted()
- Write readable and DRY (Don't Repeat Yourself) code

19. VIRTUAL ENVIRONMENTS & DEPENDENCY MANAGEMENT

- When working on different Python projects, you may need different versions of libraries. To keep things clean, we use something called a Virtual Environment — it's like a mini Python setup just for that one project!

19.1 venv Module

- Think of venv as a private sandbox for your project:

```
python -m venv myenv  # Create environment
```

- Then activate it:
- *On Windows:*

```
myenv\Scripts\activate
```

- *On Mac/Linux:*

```
source myenv/bin/activate
```

- *You'll now see (myenv) before your terminal — you're inside the virtual environment!*

19.2 requirements.txt

- This is a file that lists all the libraries your project uses.
- *Create it with:*

```
pip freeze > requirements.txt
```

- *To install everything later on another computer:*

```
pip install -r requirements.txt
```

19. VIRTUAL ENVIRONMENTS & DEPENDENCY MANAGEMENT

19.3 pip freeze

- This command lists all installed Python packages and their versions:

```
pip freeze
```

- Use it to track what your project depends on.

20. POPULAR LIBRARIES (OVERVIEW)

- Python has many helpful toolkits (called libraries) that make work faster and easier. Here are some famous ones:

20.1 Data Science

- NumPy: Fast math with numbers and big lists (arrays).
- pandas: Organize and analyze data in tables (like Excel).
- matplotlib: Make charts and graphs.
- seaborn: Fancy graphs built on top of matplotlib.

```
import pandas as pd
import matplotlib.pyplot as plt

data = [1, 2, 3, 4]
plt.plot(data)
plt.show()
```

20.2 Web Development

- Flask: Simple tool to make websites.
- Django: Big tool to make full-featured web apps (like Instagram).

```
# Flask Example
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello from Flask!"
```

20.3 Automation

- selenium: Control web browsers with Python.
- pyautogui: Control mouse, keyboard (like a robot).
- schedule: Run tasks automatically (like a clock).

```
import pyautogui
pyautogui.write('Hello!')
```

20. POPULAR LIBRARIES (OVERVIEW)

20.4 Machine Learning

- scikit-learn: For learning and prediction (basic ML).
- TensorFlow & PyTorch: Advanced tools to build smart apps that "learn" from data.

```
from sklearn.linear_model import LinearRegression
```

21. DEBUGGING & TESTING

- Debugging and testing help us find and fix mistakes in our code and make sure everything works correctly.

21.1 Using pdb

- pdb is Python's built-in debugger. It lets you pause your code and check what's going on step-by-step.

```
import pdb

x = 5
pdb.set_trace()  # Code stops here
y = x + 10
print(y)
```

- Use commands like n (next), c (continue), and p variable_name (to print) while debugging.

21.2 Unit Testing with unittest or pytest

- Tests check if parts of your code are working. A unit test checks one small part of your program.
- *Example using unittest:*

```
import unittest

def add(x, y):
    return x + y

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

unittest.main()
```

- You can also use pytest for easier syntax and better output.

22. MINI PROJECTS (PRACTICE MAKES PERFECT!)

- Build these to practice real-world Python coding:

22.1 Calculator

- Take input from the user and perform basic operations:

```
a = int(input("Enter a number: "))
b = int(input("Enter another: "))
print("Sum:", a + b)
```

22.2 To-Do List

- Use lists and functions to add/remove/display tasks.

22.3 Web Scraper

- Use requests and BeautifulSoup to extract info from websites.

```
import requests
from bs4 import BeautifulSoup

res = requests.get('https://example.com')
soup = BeautifulSoup(res.text, 'html.parser')
print(soup.title.text)
```

22.4 Weather App using API

- Use an API like OpenWeatherMap to get weather data for a city.

22.5 Alarm Clock

- Use datetime and playsound modules to create a simple alarm.

23. INTERVIEW QUESTIONS (BONUS)

23.1 Frequently Asked Python Questions

- What is Python and why is it popular?
- Explain *args and **kwargs.
- What is the difference between is and ==?
- How does memory management work in Python?
- What are Python decorators?

23.2 Code Challenges

- Reverse a string
- Check if a number is prime
- Find the factorial of a number
- Create a Fibonacci sequence
- Sort a list without using sort()

Was this post helpful ?

Follow Our 2nd Account

Follow For More

Tap Here