📘 The Essential Python Handbook

Prepared by: Shahnam Ul Hassan

Chapter 1: Python Basics

Introduction:

Python is a high-level, interpreted programming language known for its simplicity and readability. This chapter covers the basics including printing, variables, data types, operators, type conversion, and user input.

1. Printing in Python

Definition:

Printing displays output on the screen.

```
# Printing simple strings

print('Hello World!')                    # Output: Hello World!
print('Shahnam is my name!')             # Output: Shahnam is my name!

# Printing numbers as strings

print('23')                              # Numbers in quotes are treated as strings
```

2. Arithmetic Operations

Definition:

Arithmetic operations perform basic mathematical calculations like addition, subtraction, multiplication, and division.

```
# DMAS Rule: Division, Multiplication, Addition, Subtraction
print(35 + 35)          # Addition
print(35 - 30)          # Subtraction
print(30 / 5)           # Division
print(30 * 5)           # Multiplication
```

3. Variables and Assignment

Definition:

Variables store data values. The = operator assigns a value to a variable.

```
name = 'Shahnam'                    # String variable
age = 27                            # Integer variable

print('My name is:', name)
print('My age is:', age)
```

4. Data Types

Definition:

Data types specify the type of data a variable can hold.

Type	Example	Description
str	'Shahnam'	String, sequence of characters
int	27	Integer numbers
float	2.45	Decimal numbers
bool	True / False	Boolean values
NoneType	None	Represents no value

```
name1 = 'Shahnam'
age1 = 27
old = False
a = None
```

```
print(type(name1))  # <class 'str'>
print(type(age1))   # <class 'int'>
print(type(old))    # <class 'bool'>
print(type(a))      # <class 'NoneType'>
```

5. Arithmetic Operators

Definition:

Operators perform mathematical operations between numbers.

```
a = 2
b = 5

print(a + b)   # Addition
print(a - b)   # Subtraction
print(a * b)   # Multiplication
print(a / b)   # Division (float)
print(a % b)   # Modulus (remainder)
print(a ** b)  # Exponentiation (a^b)
```

6. Relational / Comparison Operators

Definition:

Comparison operators compare two values and return `True` or `False`.

```
c = 50
d = 20

print(c == d)   # Equal to
print(c != d)   # Not equal to
print(c > d)    # Greater than
print(c >= d)   # Greater than or equal
print(c < d)    # Less than
print(c <= d)   # Less than or equal
```

7. Assignment Operators

Definition:

Used to update the value of a variable.

```
num = 10

num = num + 10   # normal addition

num += 10        # shorthand addition

num *= 10        # multiply and assign

num /= 10        # divide and assign

num %= 10        # modulus and assign

num **= 10       # exponentiation and assign

print('num :', num)
```

8. Logical Operators

Definition:

Logical operators combine multiple conditions.

Operator	Meaning
`and`	True if both conditions are True
`or`	True if at least one condition is True
`not`	Reverses the result (True → False)

```
e = 50
```

```
f = 30


print(not True)        # False
print(not (e > f))     # False, because e > f is True


print(True and True)   # True
print(True or False)   # True
```

9. Type Conversion

Definition:

Changing a variable from one data type to another.

```
cc = 2
dd = 2.45
sum_val = cc + dd  # int + float → float
print(sum_val)


ee = 3.14
ee = str(ee)       # float → string
print(type(ee))    # <class 'str'>
```

10. User Input

Definition:

`input()` function is used to take input from the user. Input is always string by default.

```
name = input('Enter your name: ')
print('Welcome', name)

value = input('Enter a value: ')
print(type(value), value)
```

Multiple Inputs Example:

```
myName = input('Enter name: ')
myAge = input('Enter age: ')
myMarks = input('Enter marks: ')

print('Welcome', myName)
print('Age:', myAge)
print('Marks:', myMarks)
```

11. Practice Examples

1. **Sum of Two Numbers**

```
first = int(input('Enter first: '))
second = int(input('Enter second: '))
print('Sum =', first + second)
```

2 . Area of Square

```
side = float(input('Enter square side: '))

print('Area =', side * side)
```

3. Average of Two Numbers

```
firstNumber = float(input('Enter first number: '))

secondNumber = float(input('Enter second number: '))

print('Average =', (firstNumber + secondNumber)/2)
```

4. Compare Two Numbers

```
number1 = int(input('Enter first: '))

number2 = int(input('Enter second: '))

print(number1 >= number2)  # True if number1 >= number2
```

To understand this chapter please visit this site

https://github.com/ShahnamUlHassan/Complete-Python/tree/main/Practice%201

Chapter 2: Strings & Conditional Statements in Python

Prepared by: Shahnam Ul Hassan

1. String Basics

Definition: A string is a sequence of characters enclosed in quotes. Strings are used to store textual data in Python.

```
# Escape sequence characters
# \n = newline (moves text to next line)
# \t = tab space (adds horizontal spacing)
str_with_newline = "this is a string.\nwe are creating it in python"
str_with_tab = "this is a string.\twe are creating it in python"

print(str_with_newline)   # Prints string with newline
print(str_with_tab)       # Prints string with tab space
```

2. String Operations

2.1 Concatenation

Definition: Joining two or more strings together.

```
str1 = 'Shahnam'
str2 = 'moon'
print(str1 + str2)   # Output: Shahnammoon
```

2.2 Length of a String

Definition: `len()` function returns the number of characters in a string.

```
str4 = 'Hello!'
print(len(str4))     # Output: 6
```

2.3 Indexing

Definition: Access individual characters of a string using indices (starting from 0).

```
str3 = 'Shahnam'
print(str3[4])      # Output: n (5th character)
```

2.4 Slicing

Definition: Extract a portion of a string using `[start:end]`.

```
slice = 'shahnam'
```

print(slice[1:5]) # Output: 'hahn' (index 1 to 4)

2.5 Negative Indexing

Definition: Access string from the end using negative indices.

negSlice = "apple"
print(negSlice[-3:-1]) # Output: 'pl' (3rd last to 2nd last)

3. String Functions

Function	Description
`endswith()`	**Checks if string ends with given substring**
`capitalize()`	**Converts first character to uppercase**
`replace()`	**Replaces specified characters in string**
`find()`	**Returns index of first occurrence of substring**
`count()`	**Counts occurrences of a substring**

```
ends = 'ShahnamUlHassan'

print(ends.endswith('san'))   # True
```

```
cap = 'shahnam'
```

```
print(cap.capitalize())       # Shahnam


rep = 'helo'

print(rep.replace('o', 'p'))  # help


find = 'hello !'

print(find.find('l'))         # 2


count = 'hello world'

print(count.count('lo'))      # 1
```

4. Practice Questions (Strings)

1. **Find length of your name**

```
name = input('enter your name: ')

print('length of your name is : ', len(name))
```

2. Count $ symbols in a string

```
prac = 'Hi $ im $ symbol '

print(prac.count('$'))        # Output: 2
```

5. Conditional Statements

Definition: Conditional statements allow executing code based on certain conditions.

Example: Traffic Light System

```
light = 'green'


if(light == 'red'):
    print('stop')
elif(light == 'green'):
    print('go')
elif(light == 'yellow'):
    print('look')
else:
    print('light is broken')
```

6. Nested Conditionals

Definition: Placing an `if` statement inside another `if` statement.

```
age = 34
if age >= 18:          # Adult
    if age >= 80:      # Too old
        print('cannot drive')
    else:
```

```
    print('can drive')
else:               # Underage
    print('cannot drive')
```

7. Practice Problems (Conditionals)

1. **Even or Odd**

```
number = int(input('enter a number: '))
if(number % 2 == 0):
    print('EVEN')
else:
    print('ODD')
```

2. Largest of three numbers

```
a = int(input('enter first number: '))
b = int(input('enter second number: '))
c = int(input('enter third number: '))

if a >= b and a >= c:
    print('first number is largest', a)
elif b >= c:
    print('second number is largest', b)
else:
    print('third number is largest', c)
```

3. Multiple of 5

```
x = int(input('enter number: '))

if(x % 5 == 0):

    print('multiple of 5')

else:

    print('not a multiple')
```

✅ **Notes:**

- Strings are essential for storing text in Python.
- Conditional statements help control the flow of the program.
- Practice problems reinforce learning and problem-solving skills.

To understand this chapter please this site

https://github.com/ShahnamUlHassan/Complete-Python/tree/main/Practice%202

Chapter 3: Lists and Tuples

Prepared by: Shahnam Ul Hassan

1. Lists in Python

Definition: A list is an ordered collection of items which can contain elements of different data types. Lists are mutable (their elements can be changed).

```
marks = [98, 87, 78, 89, 50]  # list of marks
print(marks)                  # prints the entire list
print(type(marks))            # check the type → <class 'list'>
print(len(marks))             # total number of elements in the list
print(marks[0])               # prints first element (98)
print(marks[1])               # prints second element (87)
```

Example of a list with mixed datatypes:

```
student = ['shahnam', 98, 'lahore']
print(student)
student[0] = 'moon'           # updating first element
print(student)
```

Slicing a list: Extract part of a list using index ranges.

```
marksli = [98, 89, 87, 78, 76, 67]

print(marksli[1:4])   # prints elements at index 1,2,3 → [89,87,78]
```

2. List Methods

Append: Add element at the end of the list.

```
list = [2, 1, 3]

list.append(4)

print(list)  # [2, 1, 3, 4]
```

Sort: Arrange elements in ascending or descending order.

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

sortList.sort()    # ascending order

print(sortList)    # [1, 2, 3, 4]


reversed = [2, 1, 4, 3]

reversed.append(5)            # add element 5

reversed.sort(reverse=True)    # descending order

print(reversed)               # [5, 4, 3, 2, 1]
```

Insert: Add element at a specific index.

```
listInserted = [2, 1, 3]
```

```
listInserted.insert(1, 5)  # insert 5 at index 1
print(listInserted)        # [2, 5, 1, 3]
```

Pop: Remove element by index.

```
removed = [2, 1, 3, 1]
removed.pop(2)  # removes element at index 2
print(removed)  # [2, 1, 1]
```

3. Tuples in Python

Definition: Tuples are ordered collections of items, **immutable** (cannot be changed after creation).

```
tup = (2, 1, 3, 1)
print(type(tup))  # <class 'tuple'>
print(tup[0])     # 2
print(tup[1])     # 1
```

Tuple Methods:

```
element = (1, 2, 3, 4)
print(element.index(2))  # index of first occurrence of 2 → 1

tupCount = (1, 2, 3, 4)
print(tupCount.count(2))  # count how many times 2 appears → 1
```

4. Practice Examples

Practice 1 – Add 3 movies entered by user:

```
movies = []
```

```
mov1 = input('enter 1st movie: ')

mov2 = input('enter 2nd movie: ')

mov3 = input('enter 3rd movie: ')


movies.append(mov1)

movies.append(mov2)

movies.append(mov3)


print(movies)
```

Practice 2 – Palindrome check:

```
list1 = ['m', 'a', 'a', 'm']

copy_list1 = list1.copy()   # make a copy of list

copy_list1.reverse()        # reverse the copy


if copy_list1 == list1:     # check if reversed list is same as original

    print('palindrome')

else:

    print('not palindrome')
```

Practice 3 – Count occurrences in a list:

```
grade = ['A', 'D', 'A', 'B', 'C']

print(grade.count('A'))  # counts how many times 'A' appears → 2
```

Practice 4 – Sort grades:

```
grade1 = ['D', 'A', 'B', 'D', 'A']

grade1.sort()  # ascending order

print(grade1)  # ['A', 'A', 'B', 'D', 'D']
```

✅ **Notes:**

- Lists are mutable, tuples are immutable.
- Use `.append()`, `.insert()`, `.pop()`, `.sort()` for list operations.
- Use `.index()` and `.count()` for tuples.
- Practice examples help strengthen logical thinking and Python basics.

To understand this chapter please visit this site

https://github.com/ShahnamUlHassan/Complete-Python/tree/main/Practice%203

Chapter 4: Dictionary & Set in Python

Prepared by: Shahnam Ul Hassan

1. Dictionary in Python

Definition: A dictionary in Python is a collection of key-value pairs, where each key is unique. It allows fast access to values using keys.

```
# Dictionary with different datatypes

info = {

    'name': 'shahnam',                  # key: 'name', value: string

    'subjects': ['python', 'java', 'c++'], # value as list

    'topics': ('dict', 'set'),          # value as tuple

    'age': 27,                          # value as integer

    'id-adult':'True',                  # value as string representing boolean

}


print(info)                 # prints the whole dictionary
```

```
print(info['name'])        # accessing value using key
```

Notes:

- Dictionary keys must be immutable (string, number, tuple).
- Values can be of any datatype (string, list, tuple, int, boolean, etc.).

2. Creating and Updating Dictionary

```
# Empty dictionary and adding key-value pair
null_dictionary = {}
null_dictionary['name'] = 'shahnam'
print(null_dictionary)


# Nested dictionary (dictionary inside dictionary)
students = {
    'name': 'shahnam',
    'subjects': {
        'phy': 97,
        'bio': 67,
        'eng': 87
    }
}
print(students['subjects']) # prints inner dictionary
```

Notes:

- You can nest dictionaries to represent structured data like student marks.

3. Dictionary Methods

Method	Description
`keys()`	Returns all keys in the dictionary
`values()`	Returns all values in the dictionary
`items()`	Returns key-value pairs as tuples
`get(key)`	Returns value for given key (safe access)
`update()`	Adds or updates key-value pairs

```
student = {'name': 'shahnam', 'subject': {'phy': 97, 'bio': 67, 'eng': 87}}

print(student.keys())       # prints all keys
print(student.values())     # prints all values
print(list(student.items()))# prints all key-value pairs as list of tuples
print(student.get('name'))  # safe access
student.update({'city':'lahore'})  # add new key-value
```

```
print(student)
```

4. Set in Python

Definition: A set is an unordered collection of unique elements.

```
# Creating a set

collection = {1, 2, 4, 6}

print(collection)

print(len(collection))  # number of elements


# Empty set

a = set()


# Adding elements

addMethod = set()

addMethod.add(1)

addMethod.add(2)

addMethod.add(3)

print(addMethod)
```

Notes:

- Sets automatically remove duplicate elements.
- Use `set()` to create an empty set; `{}` creates an empty dictionary.

5. Set Methods

Method	Description
`remove(element)`	Removes a specific element
`clear()`	Removes all elements
`pop()`	Removes and returns a random element
`union(set2)`	Combines two sets
`intersection(set2)`	Returns common elements

```
removeMethod = {1, 2}
removeMethod.remove(2)
print(removeMethod)

collection.clear()  # removes all elements
print(collection)

set1 = {1, 2, 3}
set2 = {2, 3, 4}
print(set1.union(set2))        # {1,2,3,4}
print(set1.intersection(set2)) # {2,3}
```

6. Practice Examples

```
# Dictionary with multiple meanings
dictionary = {'cat': 'a small animal', 'table': ['furniture', 'list of facts']}
print(dictionary)

# Set automatically removes duplicates
subjects = {'python', 'java', 'js', 'python', 'c++'}
print(subjects)

# Taking marks input and storing in dictionary
marks = {}
marks.update({'phy': int(input('Enter phy: '))})
marks.update({'math': int(input('Enter math: '))})
marks.update({'chem': int(input('Enter chem: '))})
print(marks)

# Set with float & int values
values = {9, 9.5, 8, 8.5}
print(values)

# Set of tuples (immutable inside set)
values = {('float', 9.5), ('int', 9)}
print(values)
```

✅ **Notes for students:**

- Dictionary keys are unique and immutable.
- Values can be any datatype.
- Sets are unordered, unique, and useful for removing duplicates.
- Nested dictionaries and sets can help organize complex data efficiently.

To understand this chapter please visit this site

https://github.com/ShahnamUlHassan/Complete-Python/tree/main/Practice%204

Chapter 5: Loops in Python

Prepared by: Shahnam Ul Hassan

1. While Loop

- **Definition:** A `while` loop repeatedly executes a block of code as long as a given condition is `True`.

```
while condition:
    # code to execute
```

Examples

```
count = 1  # Loop control variable
while count <= 5:
    print('hello')
    count += 1  # increment by 1
```

```
print(count)  # prints 6 after loop ends
```

.Reverse While Loop

```
i = 5

while i >= 1:  # runs from 5 down to 1
    print(i)
    i -= 1

print('loop ended')
```

Practice Examples

1. Print numbers 1 to 100
2. Print numbers 100 to 1
3. Multiplication table of a number
4. Traverse list using `while` loop
5. Search an element in list
6. Use `break` to exit loop early
7. Use `continue` to skip iteration (e.g., print even numbers only)

2. For Loop

- **Definition:** A `for` loop is used for iterating over sequences (like lists, tuples, strings, or ranges).

Examples

```
# Loop over list
nums = [1, 2, 3, 4, 5]
for val in nums:
    print(val)

# Loop over tuple
tup = (1, 2, 3, 4, 5)
for num in tup:
    print(num)

# Loop over string
string = 'Shahnam'
for char in string:
    if char == 'n':
        print('n found')
        break
```

Range Function

- `range(stop)`: generates numbers from 0 to stop-1
- `range(start, stop)`: generates numbers from start to stop-1
- `range(start, stop, step)`: generates numbers from start to stop-1 with step

```
for i in range(2, 10, 2):  # even numbers
    print(i)
```

3. Common Loop Practice Problems

1. Print numbers from 1 to 100
2. Print numbers from 100 to 1
3. Multiplication table of a number
4. Placeholder using `pass` statement
5. Sum of first `n` numbers using `for` or `while` loop
6. Factorial of a number using `for` or `while` loop

```
# Factorial example using for loop
n = 5
fact = 1
for i in range(1, n+1):
    fact *= i
print('factorial =', fact)
```

Notes

- **Break Statement:** Immediately stops the loop.
- **Continue Statement:** Skips current iteration and moves to next.
- **Pass Statement:** Placeholder; does nothing but keeps loop syntactically correct.

To understand this chapter please visit this site

https://github.com/ShahnamUlHassan/Complete-Python/tree/main/Practice%205

Chapter 6: Functions in Python

Prepared by: Shahnam Ul Hassan

1. Functions

- **Definition:** A function is a reusable block of code that performs a specific task. Functions can take inputs (parameters), perform operations, and optionally return a value.

- **Syntax:**

```
def function_name(parameters):
    # code block
    return result   # optional
```

Examples

```
def calc_sum(a, b):
    sum_val = a + b  # local variable
    print(sum_val)
    return sum_val

calc_sum(5, 10)  # function call
```

.Function Without Parameters

```
def print_hello():
    print('hello')

print_hello()
```

.Function with Multiple Parameters

```
def calc_avg(a, b, c):
    avg = (a + b + c) / 3
    print(avg)
    return avg

calc_avg(1, 2, 3)
```

2. Built-in Functions

- `print()`: prints output; `end` parameter controls what comes after the print.
- `len()`: returns length of string/list/tuple.
- `type()`: returns the data type of a variable.
- `range()`: generates a sequence of numbers.

```
print('Python', end='$')
print('Python')        # default end='\n'


print(len("Python"))   # 6
print(type(10))        # int


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

3. User-defined Functions

- Functions can have default parameter values.

```
def calc_prod(a=2, b=4):
    result = a * b
    print(result)
    return result


calc_prod()       # uses default → 2*4 = 8
calc_prod(3, 5)   # override → 3*5 = 15
```

4. Practice Examples

1. **Print Length of List**

```
def print_len(lst):
    print(len(lst))


cities = ['lahore', 'karachi', 'faisalabad', 'islamabad']
heroes = ['abbu-bakkar', 'umar', 'usman', 'ali']


print_len(cities)  # 4
print_len(heroes)  # 4
```

2 Factorial Using Loop

```
def cal_fact(n):
    fact = 1
    for i in range(1, n+1):
        fact *= i
    print(fact)
    return fact


cal_fact(6)  # 720
```

3 Currency Converter

```
def converter(usd_val):
    pkr_val = usd_val * 253.53
```

```
    print(usd_val, 'USD =', pkr_val, 'PKR')
    return pkr_val

converter(2)  # 2 USD = 507.06 PKR
```

5. Recursion

- **Definition:** Recursion is a function calling itself to solve a problem until a base condition is met.

Examples

1. **Recursive Countdown**

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

show(3)  # prints 3, 2, 1
```

2. Factorial Using Recursion

```
def fact(n):
    if n == 0 or n == 1:
        return 1
```

```
    return fact(n-1) * n


print(fact(4))  # 24
```

3. Sum from 1 to n Using Recursion

```
def calc_sum(n):
    if n == 0:
        return 0
    return calc_sum(n-1) + n


total = calc_sum(5)  # 15
print(total)
```

4. Print List Using Recursion

```
def print_list(lst, idx=0):
    if idx == len(lst):
        return
    print(lst[idx])
    print_list(lst, idx+1)


fruits = ['mango', 'litchi', 'banana', 'apple']
print_list(fruits)
```

Notes

- Always define a **base case** for recursion to avoid infinite loops.
- Default parameters can be overridden during function call.
- Functions enhance code reusability and readability.
- Use descriptive function and variable names for clarity.

To understand this chapter please visit this
https://github.com/ShahnamUlHassan/Complete-Python/tree/main/Practice%206

📌 Chapter 7: File I/O in Python

Prepared by: Shahnam Ul Hassan

Source & Examples Reference:
You can explore more examples and detailed understanding from GitHub:

https://github.com/ShahnamUlHassan

For understanding this chapter, you will have the need of GitHub.

1. Types of Files

1. **Text files:** `.txt`, `.docx`, `.log`
2. **Binary files:** `.mp4`, `.mov`, `.png`, `.jpeg`

2. File Operations

🔹 **Open, Read & Close File**

```
f = open('read.txt', 'r')     # open in read mode
data = f.read()               # read entire content
print(data)
f.close()                     # always close file
```

🔹 **File Modes**

Mode	Meaning
`r`	Read (default)

`w`	Write (overwrite)
`x`	Create new file
`a`	Append
`b`	Binary
`t`	Text (default)
`+`	Read & write

🔹 **Reading**

```
f = open('read.txt', 'r')
data = f.read(5)        # read first 5 chars
print(data)
f.close()
```

```
f = open('read.txt', 'r')
line1 = f.readline()    # read line by line
line2 = f.readline()
f.close()
```

🔹 **Writing & Appending**

```
# Writing (overwrite)
f = open('write.txt', 'w')
f.write('I want to learn JS')
f.close()

# Appending
f = open('append.txt', 'a')
f.write(' then move on to ReactJS')
f.close()
```

🔹 Combined Modes

Mode	Behavior
`r+`	Read & write
`w+`	Write & read
`a+`	Append & read

🔹 Using `with` (Best Practice)

```
with open('demo.txt', 'r') as f:
    data = f.read()  # file auto-closed
```

```
with open('demo.txt', 'w') as f:
    f.write('new data')
```

🔹 Delete a File

```
import os
os.remove('remove.txt')  # permanently deletes file
```

3. Practice Examples

🔹 Practice 1: Write multiple lines

```
with open('practice.txt', 'w') as f:
    f.write('hi everyone\nwe are learning file I/O\n')
    f.write('using java\ni like programming in java')
```

🔹 Practice 2: Replace 'java' with 'python'

```
with open('practice.txt', 'r') as f:
    data = f.read()
    new_data = data.replace('java', 'python')

with open('practice.txt', 'w') as f:
    f.write(new_data)
```

🔹 Practice 3: Search for a word

```
def check_for_word():
```

```
    word = 'learning'
    with open('practice.txt', 'r') as f:
        data = f.read()
        if data.find(word) != -1:
            print('found')
        else:
            print('not found')

check_for_word()
```

🔹 Practice 4: Find word's line number

```
def check_for_line():
    word = 'learning'
    line_no = 1
    with open('practice.txt', 'r') as f:
        data = True
        while data:
            data = f.readline()
            if word in data:
                print(f"Word found at line {line_no}")
                return line_no
            line_no += 1
    return -1

print(check_for_line())
```

🔹 Practice 5: Count even numbers in file

```
count = 0

with open('practice.txt', 'r') as f:
    data = f.read()
    nums = data.split(',')
    for val in nums:
        if val.isdigit() and int(val) % 2 == 0:
            count += 1


print(f"Total even numbers: {count}")
```

💡 **Notes:**

- Always use `with open(...)` for automatic file closing.
- Choose file mode carefully (`r`, `w`, `a`, `r+`, `w+`, `a+`).
- Use `.read()`, `.readline()`, `.write()` depending on your need.
- GitHub link contains additional examples and projects for hands-on practice.

📌 Chapter 8: Object-Oriented Programming (OOP) in Python

Prepared by: Shahnam Ul Hassan

This chapter covers **classes, objects, constructors, attributes, methods**, and key OOP principles such as **abstraction, encapsulation, and static methods**.

1️⃣ Class & Object

```
class Student:

    # Class attribute

    name = 'Shahnam'


# Object (instance of class)

s1 = Student()

print(s1.name)  # Access class attribute
```

Class → blueprint for objects

Object → instance of a class

Class attributes → shared across all objects

2️⃣ Constructor (`__init__`)

```
class Student:

    def __init__(self, name, marks):

        self.name = name        # Instance attribute

        self.marks = marks

        print('Adding a new student in database..')
```

```
s1 = Student('Shahnam', 97)

print(s1.name, s1.marks)
```

3️⃣ Class & Instance Attributes

```
class Student:

    college_name = 'abc college'  # Class attribute


    def __init__(self, name, marks):

        self.name = name

        self.marks = marks


s1 = Student('Shahnam', 97)

s2 = Student('Hassan', 88)

print(s2.college_name)     # Access class attribute via object

print(Student.college_name)  # Access via class
```

4️⃣ Methods in Class

```
class Student:

    college_name = 'abcd college'


    def __init__(self, name, marks):

        self.name = name

        self.marks = marks
```

```
    def welcome(self):
        print('Welcome student,', self.name)

    def get_marks(self):
        return self.marks

s1 = Student('moon', 97)
s1.welcome()            # Instance method call
print(s1.get_marks())
```

5️⃣ Practice: Average Marks

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

    def get_avg(self):
        total = sum(self.marks)
        print('Hi', self.name, 'your avg score is', total/len(self.marks))

s1 = Student('John', [99, 98, 97])
s1.get_avg()
```

6️⃣ Static Methods

```
class Student:
    @staticmethod
    def get_avg(name, marks):
        total = sum(marks)
        print('Hi', name, 'your avg score is', total/len(marks))

Student.get_avg('John', [99, 98, 97])  # Called via class
```

- Static methods do not require `self`
- Useful for utility functions

7️⃣ Abstraction Example

```
class Car:
    def __init__(self):
        self.acc = False
        self.brk = False
        self.clutch = False

    def start(self):
        self.clutch = True
        self.acc = True
        print('Car started ..')

car1 = Car()
car1.start()
```

Users interact with **start()**, internal details are hidden

8 Encapsulation Example

```
class Car:
    def __init__(self):
        self.__acc = False
        self.__brk = False
        self.__clutch = False

    def start(self):
        self.__clutch = True
        self.__acc = True
        print('Car started...')

    def get_acc(self):
        return self.__acc

    def apply_brake(self):
        self.__brk = True
        self.__acc = False
        print("Brake applied, car stopped.")

car1 = Car()
car1.start()
print("Accelerator:", car1.get_acc())
```

car1.apply_brake()

Private attributes → `__acc`, `__brk` (cannot access directly)

Access via **getter/setter methods**

9 Practice: Bank Account

```
class Account:
    def __init__(self, bal, acc):
        self.balance = bal
        self.account_no = acc

    def debit(self, amount):
        self.balance -= amount
        print('Rs.', amount, "was debited")

    def credit(self, amount):
        self.balance += amount
        print('Rs.', amount, 'was credited')

    def get_balance(self):
        return self.balance

acc1 = Account(10000, 12345)
acc1.debit(1000)
acc1.credit(500)
```

Encapsulates money transactions

Each account is an object

This chapter summary with examples makes it easy to **add your own examples** by following the link.

https://github.com/ShahnamUlHassan/Complete-Python/tree/main/Practice%208

📌 OOP in Python (Part 2) – Notes

Prepared by: Shahnam Ul Hassan

1️⃣ `del` Keyword

Used to delete objects or variables.

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

s1 = Student('Hassan')
print(s1.name)
del s1  # deletes the object
```

2️⃣ Private-like Attributes & Methods

- Python uses `__` to indicate private attributes/methods (name mangling).

```
class Person:
    __name = 'abc'   # private attribute

# print(Person().__name) → will raise error

class Person:
    __name = 'abc'
    def __hello(self):
        print('hello user')
    def welcome(self):
        self.__hello()

p1 = Person()
p1.welcome()  # call public method which calls private method
```

3️⃣ Inheritance

- Mechanism for one class to use properties/methods of another.
- **Types:**
    1. Single Inheritance
    2. Multi-level Inheritance
    3. Multiple Inheritance

```
# Single Inheritance
class Car:
    @staticmethod
    def start(): print("car started")

class ToyotaCar(Car):
    def __init__(self, name): self.name = name

car1 = ToyotaCar("Fortuner")
car1.start()  # inherited method

# Multi-level Inheritance
class Fortuner(ToyotaCar):
    def __init__(self, type): self.type = type



# Multiple Inheritance
class A: varA = 'class A'
class B: varB = 'class B'
class C(A, B): varC = 'class C'

c1 = C()
print(c1.varA, c1.varB, c1.varC)
```

4 `super()` Method

- Calls parent class methods/constructors.

```
class Animal:
    def __init__(self, name): self.name = name
    def sound(self): print("Animals make sounds")

class Dog(Animal):
```

```
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed
    def sound(self):
        super().sound()
        print("Dog barks")

dog1 = Dog("Tommy", "German Shepherd")
dog1.sound()
```

5️⃣ Class Methods

- Use `@classmethod` decorator.
- Access/modify class variables or provide alternate constructors.

```
class Student:
    school_name = "ABC School"

    @classmethod
    def change_school(cls, new_name):
        cls.school_name = new_name

    @classmethod
    def from_string(cls, student_str):
        name, marks = student_str.split("-")
        return cls(name, int(marks))
```

6️⃣ Property Decorator

- Use `@property` for getter, `@<prop>.setter` for setter, `@<prop>.deleter` for deleter.

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

    @property
    def marks(self): return self._marks

    @marks.setter
    def marks(self, value):
        if 0 <= value <= 100: self._marks = value
```

```
        else: print("Invalid marks!")

    @marks.deleter
    def marks(self): del self._marks
```

7️⃣ Polymorphism

- **Method Overriding**: child class modifies parent method.
- **Operator Overloading**: `__add__`, `__sub__`, `__mul__`.
- **Duck Typing**: behavior-based, not type-based.
- **Built-in**: `len()`, `+` for strings/lists, etc.

```
# Method Overriding
class Dog(Animal):
    def speak(self): return "Woof"

# Operator Overloading
class Number:
    def __add__(self, other): return self.value + other.value

# Duck Typing
class Sparrow: def fly(self): return "Sparrow flying"
class Airplane: def fly(self): return "Airplane flying"

# Built-in polymorphism
len("Hello"), len([1,2,3]), "Hello" + "World"
```

✅ Practice Tip

- To see examples in action and add your own, clone from GitHub:

GitHub Example Reference:

https://github.com/ShahnamUlHassan/Complete-Python/tree/main/Practice%209

Acknowledgement

I would like to express my heartfelt gratitude to my esteemed mentor, [Sir's Ahmad Basit], whose invaluable guidance, constant encouragement, and expert insights have made this work possible. The understanding and knowledge I have gained throughout this project are largely thanks to their dedicated efforts, patience, and unwavering support. Without their mentorship and advice, compiling these notes and organizing the concepts in a clear and coherent manner would not have been possible. I am deeply indebted for their contribution to my learning journey and sincerely acknowledge their role in shaping this work.