TOPICS:

1. Installation & Environment Setup
2. Data Types
3. Operators
4. Control Flow (loops & conditionals)
5. Functions & Scope
6. File Handling
7. OOPs
8. Iterators, Generators, Coroutines, Decorators & Closures
9. Exceptional Handling & Logging
10. Modules, Libraries & Packages
11. Multi-threading & Multiprocessing
12. Memory Management & Optimization
13. Python Debugging, Testing & Best Practices
14. Regular Expressions
15. Database Handling (Sqlite3 & SQLAlchemy)
16. Web Scraping, Automation & API Handling(flask, django, fastapi)
17. Meta-Programming (Metaclasses & Reflection)
18. Design Patterns in Python
19. Network Programming
20. Security
21. AI-ML (Pandas)
22. AI-ML (Numpy)
23. AI-ML (PyTorch)
24. AI-ML (Tensorflow)
25. AI-ML (Data Visualization)

1. INSTALLATION & ENVIRONMENT SETUP

In [1]:
```
# to check if it is installed or not
! python --version

# to create virtual env in windows
# ! python -m venv venv

# to activate virtual env
# ! venv\Scripts\activate

# create a requirements.txt
# to install all packages in requirements.txt -> pip install -r requirements.txt
```

```
Python 3.12.3
```

2. Data Types

In [5]:
```
import pandas as pd

# create a DataFrame
df = pd.DataFrame({
    "Category":["Numeric", "Numeric", "Numeric", "Text", "Boolean", "Sequence",
    "Data Type": ["int", "float", "complex", "str", "bool", "list", "tuple", "ra
    "Example": ["x = 10", "y = 10.5", "z = 3 + 4j",
                's = "Hello"', "flag = True", "[1, 2, 3]",
                "(1, 2, 3)", "range(5)", "{1, 2, 3}",
                "frozenset([1,2,3])", '{"key": "value"}',
                "b'hello'", "bytearray(5)", "memoryview(bytes(5))", "x = None"]
})

df
```

Out[5]:

	Category	Data Type	Example
0	Numeric	int	x = 10
1	Numeric	float	y = 10.5
2	Numeric	complex	z = 3 + 4j
3	Text	str	s = "Hello"
4	Boolean	bool	flag = True
5	Sequence	list	[1, 2, 3]
6	Sequence	tuple	(1, 2, 3)
7	Sequence	range	range(5)
8	Set Types	set	{1, 2, 3}
9	Set Types	frozenset	frozenset([1,2,3])
10	Mapping	dict	{"key": "value"}
11	Binary	bytes	b'hello'
12	Binary	bytearray	bytearray(5)
13	Binary	memoryview	memoryview(bytes(5))
14	None Type	NoneType	x = None

In [ ]:
```
## NUMERIC DATA TYPES
# int methods
x = 10
x.bit_length() # 4 # number of bits required to represent the number in binary
x.to_bytes(2, byteorder='big') # b'\x00\n' # coverting to bytes
x.to_bytes(2, byteorder='little') # b'\n\x00'
x.from_bytes(b'\x00\n', byteorder='big') # 10 # coverting from bytes
x.from_bytes(b'\n\x00', byteorder='little') # 10
x.bit_count() # 2 # number of 1 bits in the binary representation of the number

# float methods
y = 10.5
y.as_integer_ratio() # (21, 2) # return a tuple of two integers whose ratio is e
y.is_integer() # False # check if the float is an integer
y.hex() # '0x1.5000000000000p+3' # hexadecimal representation of the float
```

```
fromhex = float.fromhex('0x1.5000000000000p+3') # 10.5 # convert from hexadecima
from_float = float.fromhex(y.hex()) # 10.5

# complex methods
z = 3 + 4j
z.real # 3.0 # real part of the complex number
z.imag # 4.0 # imaginary part of the complex number
z.conjugate() # (3-4j) # conjugate of the complex number
z.conjugate().imag # -4.0

# Built-in functions for numeric data types
abs(-10) # 10 # absolute value
divmod(10, 3) # (3, 1) # quotient and remainder
pow(2, 3) # 8 # power
round(10.5) # 10 # round off
round(10.5, 0) # 10.0
round(10.5, 1) # 10.5
sum([1, 2, 3]) # 6 # sum of elements
max([1, 2, 3]) # 3 # maximum element
min([1, 2, 3]) # 1 # minimum element

# math module
import math
math.ceil(10.5) # 11 # round up
math.floor(10.5) # 10 # round down
math.trunc(10.5) # 10 # truncate
math.factorial(5) # 120 # factorial
math.gcd(10, 5) # 5 # greatest common divisor
math.lcm(10, 5) # 10 # least common multiple
math.isqrt(10) # 3 # integer square root
math.sqrt(10) # 3.1622776601683795 # square root
math.exp(1) # 2.718281828459045 # exponential
math.log(10) # 2.302585092994046 # natural logarithm
math.log10(10) # 1.0 # base 10 logarithm
math.log2(10) # 3.321928094887362 # base 2 logarithm
math.isfinite(float('inf')) # False # check if the number is finite
math.isinf(float('inf')) # True # check if the number is infinite
math.isnan(float('nan')) # True # check if the number is not a number
math.isclose(10.5, 10.5000000001) # True # check if two numbers are close
math.isclose(10.5, 10.5000001) # False
math.isclose(10.5, 10.5000001, rel_tol=1e-6) # True
math.isclose(10.5, 10.5000001, rel_tol=1e-7) # False

## String Data Type
# String methods
s = "Hello"
s.capitalize() # 'Hello' # capitalize the first letter
s.casefold() # 'hello' # convert to lowercase
s.center(10, '*') # '**Hello***' # center align with padding
s.count('l') # 2 # count the number of occurrences
s.encode() # b'Hello' # encode the string
s.endswith('o') # True # check if it ends with the specified value
s.startswith('H') # True # check if it starts with the specified value
s.find('l') # 2 # find the first occurrence
s.rfind('l') # 3 # find the last occurrence
s.index('l') # 2 # find the first occurrence
s.rindex('l') # 3 # find the last occurrence
s.isalnum() # True # check if all characters are alphanumeric
s.isalpha() # True # check if all characters are alphabetic
s.isascii() # True # check if all characters are ASCII
```

```
s.isdecimal() # False # check if all characters are decimals
s.isdigit() # False # check if all characters are digits
s.isidentifier() # False # check if it is a valid identifier
s.islower() # False # check if all characters are lowercase
s.isnumeric() # False # check if all characters are numeric
s.isprintable() # True # check if all characters are printable
s.isspace() # False # check if all characters are whitespaces
s.istitle() # True # check if the string is titlecased
s.isupper() # False # check if all characters are uppercase
s.lower() # 'hello' # convert to lowercase
s.upper() # 'HELLO' # convert to uppercase
s.swapcase() # 'hELLO' # swap case
s.title() # 'Hello' # title case
s.strip() # 'Hello' # remove leading and trailing whitespaces
s.lstrip() # 'Hello ' # remove leading whitespaces
s.rstrip() # ' Hello' # remove trailing whitespaces
s.replace('l', 'L') # 'HeLLo' # replace all occurrences
s.split('l') # ['He', '', 'o'] # split the string
s.rsplit('l') # ['He', 'lo'] # split the string from the right
s.partition('l') # ('He', 'l', 'lo') # partition the string
s.rpartition('l') # ('Hel', 'l', 'o') # partition the string from the right
s.zfill(10) # '00000Hello' # zero padding
s.join(['1', '2', '3']) # '1Hello2Hello3' # join the strings
s.format() # 'Hello' # format the string

## Boolean Data Type
# Boolean methods
flag1 = True
flag2 = False
type(flag1) # bool
Value = 2
bool(Value) # True
bool(0) # False
bool(0.0) # False
bool('') # False
bool([]) # False
bool({}) # False
bool(()) # False
bool(None) # False
bool(flag1) # True
isinstance(flag1, bool) # True # check if it is a boolean
flag1 and flag2 # False # logical AND
flag1 or flag2 # True # logical OR
not flag1 # False # logical NOT
flag1 & flag2 # False # bitwise AND
flag1 | flag2 # True # bitwise OR
flag1 ^ flag2 # True # bitwise XOR
~flag1 # -2 # bitwise NOT
flag1 == flag2 # False # equality
flag1 != flag2 # True # inequality
flag1 is flag2 # False # identity
flag1 is not flag2 # True # non-identity

## Sequence Data Types
# List methods
lst = [1, 2, 3]
lst.append(4) # [1, 2, 3, 4] # append an element
lst.extend([5, 6]) # [1, 2, 3, 4, 5, 6] # extend the list
lst.insert(0, 1) # [1, 1, 2, 3, 4, 5, 6] # insert an element at the specified in
lst.remove(1) # [1, 2, 3, 4, 5, 6] # remove the first occurrence of the element
```

```
lst.pop() # [1, 2, 3, 4, 5] # remove the last element
lst.pop(0) # [2, 3, 4, 5] # remove the element at the specified index
lst.index(3) # 1 # find the index of the element
lst.count(3) # 1 # count the number of occurrences
lst.sort() # [2, 3, 4, 5] # sort the list
lst.reverse() # [5, 4, 3, 2] # reverse the list
lst.clear() # [] # clear the list
lst.copy() # [5, 4, 3, 2] # shallow copy the list
lst = [1, 2, 3, 4, 5]
lst[0] # 1 # access an element
lst[1:3] # [2, 3] # slice the list
lst[::2] # [1, 3, 5] # slice with step
lst[-1] # 5 # negative indexing
lst[-3:-1] # [3, 4] # negative slicing
lst[::-1] # [5, 4, 3, 2, 1] # reverse the list
lst + [6, 7] # [1, 2, 3, 4, 5, 6, 7] # concatenate lists
lst * 2 # [1, 2, 3, 4, 5, 1, 2, 3, 4, 5] # repeat the list
len(lst) # 5 # length of the list
min(lst) # 1 # minimum element
max(lst) # 5 # maximum element
sum(lst) # 15 # sum of elements
sorted(lst) # [1, 2, 3, 4, 5] # sorted list

# Tuple methods
tpl = (1, 2, 3)
tpl.count(1) # 1 # count the number of occurrences
tpl.index(2) # 1 # find the index of the element
tpl[0] # 1 # access an element
tpl[1:3] # (2, 3) # slice the tuple
tpl[::2] # (1, 3) # slice with step
tpl[-1] # 3 # negative indexing
tpl[-3:-1] # (1, 2) # negative slicing
tpl[::-1] # (3, 2, 1) # reverse the tuple
tpl + (4, 5) # (1, 2, 3, 4, 5) # concatenate tuples
tpl * 2 # (1, 2, 3, 1, 2, 3) # repeat the tuple
len(tpl) # 3 # length of the tuple
min(tpl) # 1 # minimum element
max(tpl) # 3 # maximum element
sum(tpl) # 6 # sum of elements
sorted(tpl) # [1, 2, 3] # sorted list

# Range methods
rng = range(1, 5, 2) # Start=1, Stop=5, Step=2
list(rng) # [1, 3] # convert to list

## Set Types
# Set methods
st = {1, 2, 3}
st.add(4) # {1, 2, 3, 4} # add an element
st.update({5, 6}) # {1, 2, 3, 4, 5, 6} # update the set
st.remove(1) # {2, 3, 4, 5, 6} # remove the element
st.discard(2) # {3, 4, 5, 6} # discard the element
st.pop() # 3 # remove and return an arbitrary element
st.clear() # set() # clear the set
st.copy() # {3, 4, 5, 6} # shallow copy the set
st = {1, 2, 3, 4, 5}
st.union({6, 7}) # {1, 2, 3, 4, 5, 6, 7} # union of sets
st.intersection({4, 5, 6}) # {4, 5} # intersection of sets
st.difference({4, 5, 6}) # {1, 2, 3} # difference of sets
st.symmetric_difference({4, 5, 6}) # {1, 2, 3, 6} # symmetric difference of sets
```

```
st.isdisjoint({6, 7}) # False # check if two sets are disjoint
st.issubset({1, 2, 3, 4, 5, 6, 7}) # True # check if a set is a subset
st.issuperset({1, 2, 3, 4, 5}) # True # check if a set is a superset
len(st) # 5 # length of the set

# Frozenset methods
fst = frozenset({1, 2, 3})
fst.copy() # frozenset({1, 2, 3}) # shallow copy the frozenset
fst.union({4, 5}) # frozenset({1, 2, 3, 4, 5}) # union of frozensets
fst.intersection({2, 3, 4}) # frozenset({2, 3}) # intersection of frozensets
fst.difference({2, 3, 4}) # frozenset({1}) # difference of frozensets
fst.symmetric_difference({2, 3, 4}) # frozenset({1, 4}) # symmetric difference o
fst.isdisjoint({4, 5}) # True # check if two frozensets are disjoint
fst.issubset({1, 2, 3, 4, 5}) # True # check if a frozenset is a subset
fst.issuperset({1, 2, 3}) # True # check if a frozenset is a superset
len(fst) # 3 # length of the frozenset

## Mapping Data Type
# Dictionary methods
dct = {"key": "value"}
dct["key"] # 'value' # access the value
dct.get("key") # 'value' # get the value
dct.keys() # dict_keys(['key']) # get the keys
dct.values() # dict_values(['value']) # get the values
dct.items() # dict_items([('key', 'value')]) # get the key-value pairs
dct.pop("key") # 'value' # remove and return the value
dct.popitem() # ('key', 'value') # remove and return the key-value pair
dct.clear() # {} # clear the dictionary
dct.copy() # {'key': 'value'} # shallow copy the dictionary
dct.update({"key": "value"}) # {'key': 'value'} # update the dictionary
dct = {"key": "value"}
dct.setdefault("key", "default") # 'value' # get the value or set the default

## Binary Data Types
# Bytes methods
b = b'hello'
b.capitalize() # b'Hello' # capitalize the first letter
b.center(10, b'*') # b'**hello***' # center align with padding
b.count(b'l') # 2 # count the number of occurrences
b.decode() # 'hello' # decode the bytes
b.endswith(b'o') # True # check if it ends with the specified value
b.startswith(b'h') # True # check if it starts with the specified value
b.find(b'l') # 2 # find the first occurrence
b.rfind(b'l') # 3 # find the last occurrence
b.index(b'l') # 2 # find the first occurrence
b.rindex(b'l') # 3 # find the last occurrence
b.isalnum() # True # check if all characters are alphanumeric
b.isalpha() # True # check if all characters are alphabetic
b.isascii() # True # check if all characters are ASCII
b.isdecimal() # False # check if all characters are decimals
b.isdigit() # False # check if all characters are digits
b.islower() # True # check if all characters are lowercase
b.isnumeric() # False # check if all characters are numeric
b.isspace() # False # check if all characters are whitespaces
b.istitle() # False # check if the string is titlecased
b.isupper() # False # check if all characters are uppercase
b.lower() # b'hello' # convert to lowercase
b.upper() # b'HELLO' # convert to uppercase
b.swapcase() # b'HELLO' # swap case
b.title() # b'Hello' # title case
```

```
b.strip() # b'hello' # remove leading and trailing whitespaces
b.lstrip() # b'hello' # remove leading whitespaces
b.rstrip() # b'hello' # remove trailing whitespaces
b.replace(b'l', b'L') # b'heLLo' # replace all occurrences
b.split(b'l') # [b'he', b'', 'o'] # split the string
b.rsplit(b'l') # [b'he', b'o'] # split the string from the right
b.partition(b'l') # (b'he', b'l', b'lo') # partition the string
b.rpartition(b'l') # (b'he', b'l', b'o') # partition the string from the right
b.zfill(10) # b'00000hello' # zero padding
b.join([b'1', b'2', b'3']) # b'1hello2hello3' # join the strings

# Bytearray methods
ba = bytearray(b'hello')
ba.capitalize() # bytearray(b'Hello') # capitalize the first letter
ba.center(10, b'*') # bytearray(b'**hello***') # center align with padding

# Memoryview methods
mv = memoryview(b'hello')
mv.obj # b'hello' # get the underlying object
mv.tobytes() # b'hello' # convert to bytes
mv.hex() # '68656c6c6f' # hexadecimal representation
mv[0] # 104 # access an element
mv[1:3] # b'el' # slice the memoryview

## None Type
# NoneType methods
x = None
type(x) # NoneType
isinstance(x, type(None)) # True
x is None # True
```

3. Operators

In [ ]:

```
## Arithmetic Operators
# addition
a = 10
b = 20
a + b # 30
# subtraction
a - b # -10
# multiplication
a * b # 200
# division
a / b # 0.5
# floor division
a // b # 0
# modulus
a % b # 10
# exponentiation
a ** b # 100000000000000000000
# negation
-a # -10
# positive
+a # 10

## Comparison Operators
# equal
a == b # False
# not equal
```

```
a != b # True
# greater than
a > b # False
# less than
a < b # True
# greater than or equal to
a >= b # False
# less than or equal to
a <= b # True

## Logical Operators
# and
True and False # False
# or
True or False # True
# not
not True # False

## Bitwise Operators
# and
a & b # 0 # 1010 & 11110
# or
a | b # 30 # 1010 | 11110
# xor
a ^ b # 30 # 1010 ^ 11110
# not
~a # -11 # ~1010
# left shift
a << 2 # 40 # 101000
# right shift
a >> 2 # 2 # 10

## Assignment Operators
# addition
a += b # a = a + b
# subtraction
a -= b # a = a - b
# multiplication
a *= b # a = a * b
# division
a /= b # a = a / b
# floor division
a //= b # a = a // b
# modulus
a %= b # a = a % b
# exponentiation
a **= b # a = a ** b
# bitwise and
a &= b # a = a & b
# bitwise or
a |= b # a = a | b
# bitwise xor
a ^= b # a = a ^ b
# bitwise left shift
a <<= b # a = a << b
# bitwise right shift
a >>= b # a = a >> b

## Membership Operators
# in
```

```
1 in [1, 2, 3] # True
# not in
4 not in [1, 2, 3] # True

## Identity Operators
# is
a is b # False
# is not
a is not b # True
```

4. Conditional Statements (Loops & conditions)

In [ ]:

```
## Control Structures
# if-elif-else
a = 10
if a > 10:
    print("Greater than 10")
elif a < 10:
    print("Less than 10")
else:
    print("Equal to 10")

# for loop
for i in range(5):
    print(i)

# while loop
i = 0
while i < 5:
    print(i)
    i += 1

# break
for i in range(5):
    if i == 3:
        break # exit the loop
    print(i)

# continue
for i in range(5):
    if i == 3:
        continue # skip the iteration
    print(i)

# pass
for i in range(5):
    pass # do nothing

# try-except
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Division by zero")

# try-except-else-finally
try:
    x = 1 / 1
except ZeroDivisionError:
    print("Division by zero")
```

```
else:
    print("No exceptions")
finally:
    print("Finally block")

# raise
try:
    raise Exception("Error")
except Exception as e:
    print(e)

# assert
x = 10
assert x == 10, "x should be 10"

# Terinary Operator
x = 10
y = 20
z = x if x > y else y
z # 20
```

In [19]:

```
## Pattern Questions
# 1. Print the following pattern
# *
# **
# ***
# ****
# *****
for i in range(1, 6):
    print("*" * i)

# 2. Print the following pattern
#     *
#    **
#   ***
#  ****
# *****
for i in range(1, 6):
    print(" " * (5 - i) + "*" * i)

# 3. Print the following pattern
#     *
#    ***
#   *****
#  *******
# *********
for i in range(1, 6):
    print(" " * (5 - i) + "*" * (2 * i - 1))

# 4. Print the following pattern
# *********
#  *******
#   *****
#    ***
#     *
for i in range(5, 0, -1):
    print(" " * (5 - i) + "*" * (2 * i - 1))

# 5. Print the following pattern
# 1
```

```
# 12
# 123
# 1234
# 12345
for i in range(1, 6):
    print("".join(str(j) for j in range(1, i + 1)))

# 6. Print the following pattern
# 1
# 22
# 333
# 4444
# 55555
for i in range(1, 6):
    print("".join(str(i) for j in range(1, i + 1)))

# 7. Print the following pattern
# 1
# 21
# 321
# 4321
# 54321
for i in range(1, 6):
    print("".join(str(j) for j in range(i, 0, -1)))

# 8. Print the following pattern (Floyd's Triangle)
# 1
# 23
# 456
# 78910
# 1112131415
n = 1
for i in range(1, 6):
    for j in range(i):
        print(n, end="")
        n += 1
    print()

# 9. Print the following pattern
# 1
# 32
# 654
# 10987
# 1514131211
n = 1
for i in range(1, 6):
    for j in range(i):
        print(n, end="")
        n -= 1
    n += 2 * i
    print()

# 10. Print the following pattern
# 1
# 121
# 12321
# 1234321
# 123454321
for i in range(1, 6):
    print("".join(str(j) for j in range(1, i + 1)) + "".join(str(j) for j in ran
```

```python
# 11. Print the following pattern
# 1
# 212
# 32123
# 4321234
# 543212345
for i in range(1, 6):
    print("".join(str(j) for j in range(i, 0, -1)) + "".join(str(j) for j in ran

# 12. Print the following pattern
# 1
# 232
# 34543
# 4567654
# 567898765
n = 1
for i in range(1, 6):
    for j in range(i):
        print(n, end="")
        n += 1
    for j in range(i - 1, 0, -1):
        print(n - 2, end="")
        n -= 1
    n += i
    print()

# 13. Print the following pattern
# 1
# 232
# 34543
# 4567654
# 567898765
n = 1
for i in range(1, 6):
    for j in range(i):
        print(n, end="")
        n += 1
    for j in range(i - 1, 0, -1):
        print(n - 2, end="")
        n -= 1
    n += i
    print()

# 14. Print the following pattern
# 1
# 121
# 12321
# 1234321
# 123454321
for i in range(1, 6):
    print("".join(str(j) for j in range(1, i + 1)) + "".join(str(j) for j in ran

# 15. Print the following pattern
# 1
# 11
# 121
# 1331
# 14641
n = 1
```

```
for i in range(1, 6):
    for j in range(i):
        if j == 0 or j == i - 1:
            print(1, end="")
        else:
            n = n * (i - j) // j
            print(n, end="")
    print()

# 16. Print the following pattern
# 1
# 11
# 21
# 1211
# 111221
import itertools
n = 1
for i in range(1, 6):
    print(n, end="")
    n = str(n)
    n = "".join(str(len(list(group))) + key for key, group in itertools.groupby(
    print()

# 17. Print the following pattern
# 1
# 11
# 12
# 1121
# 122111
n = 1
for i in range(1, 6):
    print(n, end="")
    n = str(n)
    n = "".join(str(len(list(group))) + key for key, group in itertools.groupby(
    print()

# 18. Print the following pattern
# 1
# 11
# 21
# 1211
# 111221
n = 1
for i in range(1, 6):
    print(n, end="")
    n = str(n)
    n = "".join(str(len(list(group))) + key for key, group in itertools.groupby(
    print()

# pascal triangle
n = 5
for i in range(1, n + 1):
    c = 1
    for j in range(1, i + 1):
        print(c, end=" ")
        c = c * (i - j) // j
    print()

# Diamond pattern
n = 5
```

```
for i in range(1, n + 1):
    print(" " * (n - i) + "* " * i)

for i in range(n - 1, 0, -1):
    print(" " * (n - i) + "* " * i)

# Hollow Diamond pattern
n = 5
for i in range(1, n + 1):
    if i == 1 or i == n:
        print(" " * (n - i) + "* " * i)
    else:
        print(" " * (n - i) + "* " + "  " * (i - 2) + "*")

for i in range(n - 1, 0, -1):
    if i == 1 or i == n:
        print(" " * (n - i) + "* " * i)
    else:
        print(" " * (n - i) + "* " + "  " * (i - 2) + "*")

# Square pattern
n = 5
for i in range(1, n + 1):
    print("* " * n)

# Rectangle pattern
n = 5
m = 3
for i in range(1, m + 1):
    print("* " * n)

# Hollow Square pattern
n = 5
for i in range(1, n + 1):
    if i == 1 or i == n:
        print("* " * n)
    else:
        print("* " + "  " * (n - 2) + "*")

# Hollow Rectangle pattern
n = 5
m = 3
for i in range(1, m + 1):
    if i == 1 or i == m:
        print("* " * n)
    else:
        print("* " + "  " * (n - 2) + "*")

# Cross pattern
n = 5
for i in range(1, n + 1):
    for j in range(1, n + 1):
        if j == i or j == n - i + 1:
            print("*", end="")
        else:
            print(" ", end="")
    print()

# Hollow Cross pattern
n = 5
```

```
for i in range(1, n + 1):
    for j in range(1, n + 1):
        if j == i or j == n - i + 1:
            print("*", end="")
        else:
            print(" ", end="")
    print()

# X pattern
n = 5
for i in range(1, n + 1):
    for j in range(1, n + 1):
        if j == i or j == n - i + 1:
            print("*", end="")
        else:
            print(" ", end="")
    print()
```

```
*
**
***
****
*****
    *
   **
  ***
 ****
*****
    *
   ***
  *****
 *******
*********
*********
 *******
  *****
   ***
    *
1
12
123
1234
12345
1
22
333
4444
55555
1
21
321
4321
54321
1
23
456
78910
1112131415
1
21
432
7654
1110987
1
121
12321
1234321
123454321
1
212
32123
4321234
543212345
1
343
67876
10111213121110
1516171819181716 15
```

```
1
343
67876
10111213121110
151617181918171615
1
121
12321
1234321
123454321
1
11
121
1661
12436241
1
11
21
1211
111221
1
11
21
1211
111221
1
11
21
1211
111221
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
    *
   * *
  * * *
 * * * *
* * * * *
 * * * *
  * * *
   * *
    *
    *
   * *
  *   *
 *     *
* * * * *
 *     *
  *   *
   * *
    *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

```
* * * * *
* * * * *
*       *
*       *
*       *
* * * * *
* * * * *
*       *
* * * * *
*   *
 * *
  *
 * *
*   *
*   *
 * *
  *
 * *
*   *
*   *
 * *
  *
 * *
*   *
```

5. Functions & Scope

In [ ]:
```python
## Functions
# function definition
def greet():
    print("Hello")

# function call
greet()

# function with arguments
def greet(name):
    print(f"Hello {name}")

greet("Alice")

# function with default arguments
def greet(name="Alice"):
    print(f"Hello {name}")

greet()

# function with return value
def add(a, b):
    return a + b

result = add(10, 20)

# function with multiple return values
def add_sub(a, b):
    return a + b, a - b

add, sub = add_sub(10, 20)
```

```python
# function with variable arguments
def add(*args):
    return sum(args)

result = add(1, 2, 3, 4, 5)

# function with keyword arguments
def greet(name, message):
    print(f"{message} {name}")

greet(name="Alice", message="Hello")

# function with variable keyword arguments
def greet(**kwargs):
    print(f"{kwargs['message']} {kwargs['name']}")

greet(name="Alice", message="Hello")

# lambda function
add = lambda a, b: a + b
result = add(10, 20)

# map function
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))

# filter function
numbers = [1, 2, 3, 4, 5]
even = list(filter(lambda x: x % 2 == 0, numbers))

# reduce function
from functools import reduce
numbers = [1, 2, 3, 4, 5]
sum_reduce = reduce(lambda x, y: x + y, numbers)

# recursion
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

result = factorial(5)

# functions with return type
def add(a: int, b: int) -> int:
    return a + b

result = add(10, 20)

# function scope
x = 10 # global variable
def func():
    y = 20 # local variable
    print(x, y)

func()

# global keyword
x = 10
def func():
```

```
    global x # refer to the global x
    x = 20

func()

# nonlocal keyword
def outer():
    x = 10
    def inner():
        nonlocal x # refer to the x in the outer function
        x = 20
    inner()
    print(x)

outer()

# higher order functions
def add(a, b):
    return a + b

def sub(a, b):
    return a - b

def apply(func, a, b):
    return func(a, b)

result = apply(add, 10, 20)

# function inside function
def outer():
    def inner():
        print("Inner function")
    inner()

outer()

# function as return value
def outer():
    def inner():
        print("Inner function")
    return inner

func = outer()

func()
```

```
Hello
Hello Alice
Hello Alice
Hello Alice
Hello Alice
10 20
20
Inner function
Inner function
```

6. File Handling

In [ ]:
```
## File Handling
# write to a file
```

```python
with open("file.txt", "w") as file:
    file.write("Hello") # write to the file/overwrite content

# read from a file
with open("file.txt", "r") as file:
    data = file.read() # read the entire file

# append to a file
with open("file.txt", "a") as file:
    file.write(" World") # append to the file without overwriting

# read line by line
with open("file.txt", "r") as file:
    for line in file:
        print(line)

# write line by line
with open("file.txt", "w") as file:
    file.write("Hello\n")
    file.write("World\n")

# read binary file
with open("file.txt", "rb") as file:
    data = file.read()

# write binary file
with open("file.txt", "wb") as file:
    file.write(b"Hello")

# append binary file
with open("file.txt", "ab") as file:
    file.write(b" World")

# readline
with open("file.txt", "r") as file:
    line = file.readline() # read a single line

# readlines
with open("file.txt", "r") as file:
    lines = file.readlines() # read all lines

# writelines
with open("file.txt", "w") as file:
    file.writelines(["Hello", "World"]) # write multiple lines

# seek
with open("file.txt", "r") as file:
    file.seek(5) # move the cursor to the 5th byte
    data = file.read()

# tell
with open("file.txt", "r") as file:
    file.read()
    position = file.tell() # get the current position

# flush
with open("file.txt", "w") as file:
    file.write("Hello")
    file.flush() # flush the buffer
```

```python
# close
file = open("file.txt", "w")
file.write("Hello")
file.close() # close the file

# read csv file
import csv
with open("file.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)

# write csv file
import csv
with open("file.csv", "w") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age"])
    writer.writerow(["Alice", 25])
    writer.writerow(["Bob", 30])

# read json file
import json
with open("file.json", "r") as file:
    data = json.load(file)

# write json file
import json
with open("file.json", "w") as file:
    json.dump(data, file)

# read yaml file
import yaml
with open("file.yaml", "r") as file:
    data = yaml.load(file, Loader=yaml.FullLoader)

# write yaml file
import yaml
with open("file.yaml", "w") as file:
    yaml.dump(data, file)

# read xml file
import xml.etree.ElementTree as ET
tree = ET.parse("file.xml") # parse xml file
root = tree.getroot() # get root element

# write xml file
import xml.etree.ElementTree as ET
root = ET.Element("root") # create root element
tree = ET.ElementTree(root) # create tree
tree.write("file.xml") # write to xml file

# read excel file
import pandas as pd
data = pd.read_excel("file.xlsx")

# write excel file
import pandas as pd
data.to_excel("file.xlsx")
```

7. OOPs - Object Oriencted Programming

In [ ]:
```python
## Basic OOPs Concepts
# class definition
# what is a class ?
# A class is a blueprint for creating objects (a particular data structure), pro
# what is an object ?
# An object is an instance of a class. When a class is defined, no memory is all
# what is a method ?
# A method is a function defined inside a class that operates on the object of t
# what is an attribute ?
# An attribute is a variable that is bound to an object.
class Person: # class
    def __init__(self, name, age):
        self.name = name # attribute
        self.age = age

    def greet(self): # method
        print(f"Hello {self.name}")

# class instantiation
person = Person("Alice", 25) # object

# class attributes
person.name # 'Alice' # access the attribute
person.age # 25

# class methods
person.greet()

## Inheritance
# what is inheritance?
# Inheritance is a mechanism in which one class inherits the attributes and meth
# super() method
# what is the super() method?
# The super() method is used to call the parent class constructor
class Student(Person): # child class
    def __init__(self, name, age, roll):
        super().__init__(name, age) # call the parent class constructor
        self.roll = roll

    def study(self):
        print(f"{self.name} is studying")

student = Student("Bob", 30, 101)

## Encapsulation
# what is encapsulation?
# Encapsulation is the bundling of data (attributes) and methods that operate on
# what are private attributes?
# Private attributes are attributes that are only accessible inside the class.
# what are protected attributes?
# Protected attributes are attributes that are accessible inside the class and i
class Account:
    def __init__(self, balance):
        self.__balance = balance # private attribute

    def get_balance(self):
        return self.__balance
```

```
    def set_balance(self, balance):
        self.__balance = balance

account = Account(1000)
account.get_balance() # 1000
account.set_balance(2000)
account.get_balance() # 2000

## Polymorphism
# what is polymorphism?
# Polymorphism is the ability of an object to take on many forms. The most commo

# what is method overriding? (runtime polymorphism)
# Method overriding is a feature that allows a subclass to provide a specific im
class Animal:
    def sound(self):
        return "Animals make sound"

class Dog(Animal):
    def sound(self):  # Overriding method
        return "Bark"

class Cat(Animal):
    def sound(self):  # Overriding method
        return "Meow"

dog = Dog()
cat = Cat()
print(dog.sound())  # Output: Bark
print(cat.sound())  # Output: Meow

# what is method overloading? (compile-time polymorphism)
# Method overloading is a feature that allows a class to have more than one meth
# we do method overloading using *args and **kwargs
class MathOperations:
    def add_op(self, *args):
        i = 0
        arg = 0
        while i < len(args):
            # print(f"{i}",args[i])
            arg += args[i]
            i += 1
        return arg

math_op = MathOperations()
print(math_op.add_op(5))         # Output: 5
print(math_op.add_op(5, 10))     # Output: 15
print(math_op.add_op(5, 10, 15)) # Output: 30

# polymorphism with functions & classes
class Dog:
    def sound(self):
        return "Bark"

class Cat:
    def sound(self):
        return "Meow"

def make_sound(animal):
```

```
    print(animal.sound())

dog = Dog()
cat = Cat()

make_sound(dog)  # Output: Bark
make_sound(cat)  # Output: Meow


# polymorphism using inheritance
class Vehicle:
    def fuel_type(self):
        return "Unknown"

class PetrolCar(Vehicle):
    def fuel_type(self):
        return "Petrol"

class ElectricCar(Vehicle):
    def fuel_type(self):
        return "Electric"

vehicles = [PetrolCar(), ElectricCar()]

for vehicle in vehicles:
    print(vehicle.fuel_type())

## self and __init__
# what is self?
# self represents the instance of the class. By using the "self" keyword we can
# what is __init__?
# __init__ is a special method in Python classes. It is called a constructor in
class Car:
    def __init__(self, brand, model): # __init__ constructor
        self.brand = brand # self refers to the instance of the class
        self.model = model # self.brand and self.model are attributes

    def display(self):
        print(f"Car: {self.brand} {self.model}")

car1 = Car("Toyota", "Corolla")
car1.display()
```

```
Hello Alice
Bark
Meow
5
15
30
Bark
Meow
Petrol
Electric
Car: Toyota Corolla
```

In [ ]:

```
## Intermediate OOPs Concepts
# class attributes
# what are class attributes?
# Class attributes are attributes that are shared by all instances of a class.
class Car:
```

```
    wheels = 4 # class attribute

    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def display(self):
        print(f"Car: {self.brand} {self.model}")
        print(f"Wheels: {self.wheels}")

car1 = Car("Toyota", "Corolla")
car1.display()

# instance attributes
# what are instance attributes?
# Instance attributes are attributes that are unique to each instance of a class
class Car:
    def __init__(self, brand, model):
        self.brand = brand # instance attribute
        self.model = model # instance attribute

    def display(self):
        print(f"Car: {self.brand} {self.model}")

car1 = Car("Toyota", "Corolla")
car1.display()

# instance vs class variables
# what is the difference between instance and class variables?
# Instance variables are unique to each instance of a class whereas class variab
class Car:
    wheels = 4 # class variable

    def __init__(self, brand, model):
        self.brand = brand # instance variable
        self.model = model # instance variable

    def display(self):
        print(f"Car: {self.brand} {self.model}")
        print(f"Wheels: {self.wheels}")

car1 = Car("Toyota", "Corolla")
car1.display()

# class methods vs instance methods vs static methods
# what is the difference between class methods, instance methods, and static met
# Class methods are methods that are bound to the class and not the instance of
# Instance methods are methods that are bound to the instance of the class. Work
# Static methods are methods that are not bound to the instance or class. Doesn't

class Car:
    # Class variable (common for all instances)
    total_cars = 0

    def __init__(self, brand, model, price):
        self.brand = brand  # Instance variable
        self.model = model  # Instance variable
        self.price = price  # Instance variable
        Car.total_cars += 1  # Modify class variable
```

```
    # Instance Method (works with instance attributes)
    def get_details(self):
        return f"Car: {self.brand} {self.model}, Price: ${self.price}"

    # Class Method (works with class variables)
    @classmethod
    def get_total_cars(cls):
        return f"Total cars created: {cls.total_cars}"

    # Static Method (utility function that doesn't depend on class or instance)
    @staticmethod
    def is_luxury(price):
        return price > 50000  # A car is considered luxury if price > $50,000


# ✅ Creating instances
car1 = Car("BMW", "X5", 70000)
car2 = Car("Toyota", "Camry", 35000)

# ✅ Calling instance method
print(car1.get_details())  # Output: Car: BMW X5, Price: $70000

# ✅ Calling class method
print(Car.get_total_cars())  # Output: Total cars created: 2

# ✅ Calling static method
print(Car.is_luxury(70000))  # Output: True
print(Car.is_luxury(35000))  # Output: False
```

In [ ]:

```
## Advanced OOPs Concepts
# what is composition?
# Composition is a design technique in object-oriented programming to implement
# what is aggregation?
# Aggregation is a design technique in object-oriented programming to implement
# what is association?
# Association is a relationship between two classes that establishes through the
# what is dependency?
# Dependency is a relationship between two classes where one class depends on an

## magic methods
# what are magic methods?
# Magic methods are special methods in Python that start and end with double und
# what is __str__?
# __str__ is a magic method that returns a string representation of an object.
# what is __repr__?
# __repr__ is a magic method that returns an unambiguous string representation o
# what is __add__?
# __add__ is a magic method that defines the behavior of the + operator.
# what is __sub__?
# __sub__ is a magic method that defines the behavior of the - operator.
# what is __mul__?
# __mul__ is a magic method that defines the behavior of the * operator.
# what is __truediv__?
# __truediv__ is a magic method that defines the behavior of the / operator.
# what is __floordiv__?
# __floordiv__ is a magic method that defines the behavior of the // operator.
# what is __mod__?
# __mod__ is a magic method that defines the behavior of the % operator.
# what is __pow__?
# __pow__ is a magic method that defines the behavior of the ** operator.
```

```
# what is __eq__?
# __eq__ is a magic method that defines the behavior of the == operator.
# what is __ne__?
# __ne__ is a magic method that defines the behavior of the != operator.
# what is __lt__?
# __lt__ is a magic method that defines the behavior of the < operator.
# what is __le__?
# __le__ is a magic method that defines the behavior of the <= operator.
# what is __gt__?
# __gt__ is a magic method that defines the behavior of the > operator.
# what is __ge__?
# __ge__ is a magic method that defines the behavior of the >= operator.
# what is __len__?
# __len__ is a magic method that returns the length of an object.
# what is __getitem__?
# __getitem__ is a magic method that defines the behavior of the [] operator.
# what is __setitem__?
# __setitem__ is a magic method that defines the behavior of the []= operator.
# what is __delitem__?
# __delitem__ is a magic method that defines the behavior of the del operator.
# what is __iter__?
# __iter__ is a magic method that returns an iterator object.
# what is __next__?
# __next__ is a magic method that returns the next item in an iterator.
# what is __contains__?
# __contains__ is a magic method that defines the behavior of the in operator.
# what is __call__?
# __call__ is a magic method that enables an object to be called like a function
# what is __enter__?
# __enter__ is a magic method that defines the behavior when entering a context
# what is __exit__?
# __exit__ is a magic method that defines the behavior when exiting a context ma
# what is __getattr__?
# __getattr__ is a magic method that is called when an attribute is not found in
# what is __setattr__?
# __setattr__ is a magic method that is called when an attribute is set.
# what is __delattr__?
# __delattr__ is a magic method that is called when an attribute is deleted.
# what is __dir__?
# __dir__ is a magic method that returns the list of attributes of an object.
# what is __doc__?
# __doc__ is a magic method that returns the docstring of a class or function.
# what is __class__?
# __class__ is a magic method that returns the class of an object.
# what is __bases__?
# __bases__ is a magic method that returns the base classes of a class.
# what is __subclasses__?
# __subclasses__ is a magic method that returns the subclasses of a class.
# what is __mro__?
# __mro__ is a magic method that returns the method resolution order of a class.
# what is __name__?
# __name__ is a magic method that returns the name of a class or function.
# what is __qualname__?
# __qualname__ is a magic method that returns the qualified name of a class or f
# what is __module__?
# __module__ is a magic method that returns the module in which a class or funct
# what is __dict__?
# __dict__ is a magic method that returns the dictionary containing the attribut
# what is __slots__?
# __slots__ is a magic method that restricts the attributes of a class to a pred
```

```python
# what is __hash__?
# __hash__ is a magic method that returns the hash value of an object.
# what is __bytes__?
# __bytes__ is a magic method that returns the bytes representation of an object
# what is __format__?
# __format__ is a magic method that returns a formatted string representation of
# what is __index__?
# __index__ is a magic method that returns the integer value of an object.
# what is __copy__?
# __copy__ is a magic method that returns a shallow copy of an object.
# what is __deepcopy__?
# __deepcopy__ is a magic method that returns a deep copy of an object.
# what is __reduce__?
# __reduce__ is a magic method that returns a tuple of the object's state.
# what is __reduce_ex__?
# __reduce_ex__ is a magic method that returns a tuple of the object's state wit
# what is __getstate__?
# __getstate__ is a magic method that returns the object's state.
# what is __setstate__?
# __setstate__ is a magic method that sets the object's state.
# what is __sizeof__?
# __sizeof__ is a magic method that returns the size of an object in bytes.
import copy

class MagicMethodsDemo:
    __slots__ = ('name', 'age', 'index', 'new_attr')  # Restricts attributes to

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

    # ✅ __str__ & __repr__ (String Representation)
    def __str__(self):
        return f"Person(Name: {self.name}, Age: {self.age})"

    def __repr__(self):
        return f"MagicMethodsDemo({self.name!r}, {self.age!r})"

    # ✅ Arithmetic Operators Overloading
    def __add__(self, other):
        return MagicMethodsDemo(self.name + other.name, self.age + other.age)

    def __sub__(self, other):
        return MagicMethodsDemo(self.name, self.age - other.age)

    def __mul__(self, num):
        return MagicMethodsDemo(self.name * num, self.age * num)

    def __truediv__(self, num):
        return MagicMethodsDemo(self.name, self.age / num)

    def __floordiv__(self, num):
        return MagicMethodsDemo(self.name, self.age // num)

    def __mod__(self, num):
        return MagicMethodsDemo(self.name, self.age % num)

    def __pow__(self, num):
        return MagicMethodsDemo(self.name, self.age ** num)
```

```python
    # ✅ Comparison Operators Overloading
    def __eq__(self, other):
        return self.age == other.age

    def __ne__(self, other):
        return self.age != other.age

    def __lt__(self, other):
        return self.age < other.age

    def __le__(self, other):
        return self.age <= other.age

    def __gt__(self, other):
        return self.age > other.age

    def __ge__(self, other):
        return self.age >= other.age

    # ✅ Length and Indexing
    def __len__(self):
        return self.age

    def __getitem__(self, index):
        return self.name[index]

    def __setitem__(self, index, value):
        self.name = self.name[:index] + value + self.name[index + 1:]

    def __delitem__(self, index):
        self.name = self.name[:index] + self.name[index + 1:]

    # ✅ Iterators
    def __iter__(self):
        self.index = 0
        return self

    def __next__(self):
        if self.index < len(self.name):
            char = self.name[self.index]
            self.index += 1
            return char
        else:
            raise StopIteration

    def __contains__(self, item):
        return item in self.name

    # ✅ Callable Object
    def __call__(self):
        print(f"Calling {self.name} with age {self.age}")

    # ✅ Context Manager
    def __enter__(self):
        print(f"Entering context with {self.name}")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print(f"Exiting context with {self.name}")
```

```
    # ✅ Attribute Handling
    def __getattr__(self, attr):
        return f"Attribute {attr} not found!"

    def __setattr__(self, attr, value):
        super().__setattr__(attr, value)

    def __delattr__(self, attr):
        super().__delattr__(attr)

    # ✅ Meta Information
    def get_class_info(self):
        return {
            "class": self.__class__,
            "bases": self.__class__.__bases__,
            "mro": self.__class__.__mro__,
            "module": self.__class__.__module__,
            "dict": self.__class__.__dict__,
            "name": self.__class__.__name__,
            "qualname": self.__class__.__qualname__
        }

    # ✅ Hashing, Bytes, and Formatting
    def __hash__(self):
        return hash((self.name, self.age))

    def __bytes__(self):
        return bytes(self.name, 'utf-8')

    def __format__(self, format_spec):
        return f"Formatted: {self.name} ({self.age} years old)"

    # ✅ Index Representation
    def __index__(self):
        return self.age

    # ✅ Copying & Serialization
    def __copy__(self):
        return MagicMethodsDemo(self.name, self.age)

    def __deepcopy__(self, memodict={}):
        return MagicMethodsDemo(copy.deepcopy(self.name, memodict), copy.deepcop

    def __sizeof__(self):
        return super().__sizeof__()

# ✅ Testing the Class

p1 = MagicMethodsDemo("Alice", 25)
p2 = MagicMethodsDemo("Bob", 30)

# String representation
print(str(p1))  # __str__
print(repr(p1))  # __repr__

# Arithmetic operations
print((p1 + p2).age)  # __add__
print((p2 - p1).age)  # __sub__

# Comparison
```

```
print(p1 < p2)  # __lt__

# Length and Indexing
print(len(p1))  # __len__
print(p1[1])  # __getitem__

# Iteration
for char in p1:
    print(char, end=" ")  # __iter__, __next__
print()

# Contains
print('A' in p1)  # __contains__

# Call object
p1()  # __call__

# Context Manager
with p1 as obj:
    print(obj)

# Attribute Handling
print(p1.unknown)  # __getattr__
p1.new_attr = "Test"  # __setattr__
del p1.new_attr  # __delattr__

# Meta Information
print(p1.get_class_info())  # __class__, __bases__, __mro__, __module__, __dict_

# Hashing, Bytes, and Formatting
print(hash(p1))  # __hash__
print(bytes(p1))  # __bytes__
print(format(p1))  # __format__

# Copying & Serialization
p3 = copy.copy(p1)  # __copy__
p4 = copy.deepcopy(p1)  # __deepcopy__

# Size of object
print(p1.__sizeof__())  # __sizeof__

# Note:
# 1️⃣ String Representations → __str__, __repr__
# 2️⃣ Operator Overloading → __add__, __sub__, __mul__, __truediv__, __floordiv__
# 3️⃣ Comparison Operators → __eq__, __ne__, __lt__, __le__, __gt__, __ge__
# 4️⃣ Length & Indexing → __len__, __getitem__, __setitem__, __delitem__
# 5️⃣ Iteration → __iter__, __next__
# 6️⃣ Membership Testing → __contains__
# 7️⃣ Callable Objects → __call__
# 8️⃣ Context Manager → __enter__, __exit__
# 9️⃣ Attribute Management → __getattr__, __setattr__, __delattr__
# 🔟 Meta Information → __class__, __bases__, __mro__, __module__, __dict__, __
# 🔢 Hashing & Formatting → __hash__, __bytes__, __format__
# 🔢 Copying & Serialization → __copy__, __deepcopy__, __sizeof__

## overloading
# what is operator overloading?
# Operator overloading is a feature in Python that allows the same operator to h
class Point:
    def __init__(self, x, y):
```

```
        self.x = x
        self.y = y

    # Overloading `+` operator
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

p1 = Point(2, 3)
p2 = Point(4, 5)
p3 = p1 + p2  # Uses __add__
print(f"New Point: ({p3.x}, {p3.y})")  # Output: New Point: (6, 8)

# what is method overloading?
# Method overloading is a feature that allows a class to have more than one meth
class MathOperations:
    def add_op(self, *args):
        i = 0
        arg = 0
        while i < len(args):
            # print(f"{i}",args[i])
            arg += args[i]
            i += 1
        return arg

math = MathOperations()
print(math.add_op(5))         # Output: 5
print(math.add_op(5, 10))     # Output: 15
print(math.add_op(5, 10, 15)) # Output: 30

# what is method overriding?
# Method overriding is a feature that allows a subclass to provide a specific im
class Parent:
    def show(self):
        return "This is Parent Class"

class Child(Parent):
    def show(self):  # Overriding Parent's method
        return "This is Child Class"

c = Child()
print(c.show())  # Output: This is Child Class

# what is function overloading?
# Function overloading is a feature that allows a function to have more than one
from functools import singledispatch

@singledispatch # function overloading using singledispatch decorator
def show(value):
    return f"Default: {value}"

@show.register(int)
def _(value):
    return f"Integer: {value}"

@show.register(str)
def _(value):
    return f"String: {value}"

print(show(10))     # Output: Integer: 10
print(show("Hi"))   # Output: String: Hi
```

```
print(show(3.14))   # Output: Default: 3.14

# what is function overriding?
# Function overriding is a feature that allows a function to have more than one
class Animal:
    def sound(self):
        return "Some sound"

class Dog(Animal):
    def sound(self):  # Function overriding in subclass
        return "Bark"

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


# what is constructor overloading?
# Constructor overloading is a feature that allows a class to have more than one
class Student: # try to use *args for same
    def __init__(self, name=None, age=None):
        self.name = name if name else "Unknown"
        self.age = age if age else 18

s1 = Student()
s2 = Student("Sahil")
s3 = Student("Kaushik", 30)

print(s1.name, s1.age)  # Output: Unknown 18
print(s2.name, s2.age)  # Output: Sahil 18
print(s3.name, s3.age)  # Output: Kaushik 30

# what is constructor overriding?
# Constructor overriding is a feature that allows a subclass to provide a specif
class Parent:
    def __init__(self):
        print("Parent Constructor")

class Child(Parent):
    def __init__(self):  # Overriding Constructor
        print("Child Constructor")

c = Child()  # Output: Child Constructor (Parent's constructor is overridden)

# what is operator overriding?
# Operator overriding is a feature in Python that allows the same operator to ha
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def __gt__(self, other):  # Overriding `>` operator
        return (self.length * self.width) > (other.length * other.width)

r1 = Rectangle(10, 20)
r2 = Rectangle(5, 25)
print(r1 > r2)  # Output: True (200 > 125)

# what is method signature?
# Method signature is a combination of a method’s name and the number and types
# what is function signature?
```

```python
# Function signature is a combination of a function's name and the number and ty
def add(a: int, b: int) -> int:  # Function Signature
    return a + b

class MathOperations:
    def multiply(self, x: int, y: int) -> int:  # Method Signature
        return x * y

m = MathOperations()
print(add(5, 10))       # Output: 15
print(m.multiply(2, 3)) # Output: 6

# what is constructor signature?
# Constructor signature is a combination of a constructor's name and the number
# what is operator signature?
# Operator signature is a combination of an operator's name and the number and t
# what is method resolution order?
# Method resolution order is the order in which methods are resolved/inherited i
# C3 Linearization (Depth-First, Left-to-Right) algorithm is used to calculate t
class A:
    def show(self):
        return "A"

class B(A):
    def show(self):
        return "B"

class C(A):
    def show(self):
        return "C"

class D(B, C):  # Multiple Inheritance
    pass

d = D()
print(d.show())  # Output: B  (Because Python follows MRO: D → B → C → A)
print(D.mro())   # Output: [D, B, C, A, object]

# what is operator precedence?
# Operator precedence is the order in which operators are evaluated in an expres
# what is operator associativity?
# Operator associativity is the order in which operators of the same/equal prece
result = 2 + 3 * 4  # Multiplication (*) has higher precedence than addition (+)
print(result)  # Output: 14

# Associativity Example (Left-to-Right)
result = 10 - 5 - 2  # (10 - 5) - 2 = 3 (Left-to-right associativity)
print(result)  # Output: 3

# Precedence Order (Highest to Lowest):
# 1️⃣ () Parentheses
# 2️⃣ ** Exponentiation
# 3️⃣ * / // % Multiplication/Division
# 4️⃣ + - Addition/Subtraction
# 5️⃣ == != < > Comparison Operators
# 6️⃣ and, or, not Logical Operators

# Associativity Rules:

# Left-to-Right: +, -, *, /, //, %, &, |, ^, <<, >>
```

```python
# Right-to-Left: **, =, +=, -=, *=, /=, //=, %=, **=

# what is operator arity?
# Operator arity is the number of operands an operator takes.
x = -5    # Unary Operator (-)
y = 10 + 5  # Binary Operator (+)
z = 100 if y > 10 else 50  # Ternary Operator

print(x, y, z)  # Output: -5 15 100

## Parsing Techniques
# what is operator precedence parsing?
# Operator precedence parsing is a method of parsing expressions based on the pr
# what is operator associativity parsing?
# Operator associativity parsing is a method of parsing expressions based on the
# what is operator arity parsing?
# Operator arity parsing is a method of parsing expressions based on the arity o
# what is operator overloading parsing?
# Operator overloading parsing is a method of parsing expressions based on the o
# what is operator overriding parsing?
# Operator overriding parsing is a method of parsing expressions based on the ov
# what is function overloading parsing?
# Function overloading parsing is a method of parsing expressions based on the o
# what is function overriding parsing?
# Function overriding parsing is a method of parsing expressions based on the ov
# what is constructor overloading parsing?
# Constructor overloading parsing is a method of parsing expressions based on th
# what is constructor overriding parsing?
# Constructor overriding parsing is a method of parsing expressions based on the
# what is operator signature parsing?
# Operator signature parsing is a method of parsing expressions based on the sig
# what is method signature parsing?
# Method signature parsing is a method of parsing expressions based on the signa
# what is function signature parsing?
# Function signature parsing is a method of parsing expressions based on the sig
# what is constructor signature parsing?
# Constructor signature parsing is a method of parsing expressions based on the

from functools import singledispatch

# ✅ Operator Overloading Parsing (Custom `+` operator)
class Math:
    def __init__(self, value):
        self.value = value

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

    def __gt__(self, other):  # Operator Overriding
        return self.value > other.value

# ✅ Operator Arity Parsing (Unary, Binary, Ternary)
def arity_example(x):
    y = -x  # Unary Operator
    z = x + 5  # Binary Operator
    return z if z > 10 else 0  # Ternary Operator

# ✅ Operator Precedence & Associativity Parsing
def precedence_example():
    return 2 + 3 * 4  # Multiplication has higher precedence than addition
```

```python
# ✅ Function Overloading Parsing (Using `@singledispatch`)
@singledispatch
def show(value):
    return f"Default: {value}"

@show.register(int)
def _(value):
    return f"Integer: {value}"

@show.register(str)
def _(value):
    return f"String: {value}"

# ✅ Function Overriding Parsing (Subclass Redefinition)
class Parent:
    def show(self):
        return "Parent Class"

class Child(Parent):
    def show(self):  # Function Overriding
        return "Child Class"

# ✅ Constructor Overloading Parsing (Handling Different Arguments)
class Student:
    def __init__(self, name=None, age=None):
        self.name = name if name else "Unknown"
        self.age = age if age else 18

# ✅ Constructor Overriding Parsing (Child Class Changes Parent Constructor)
class Base:
    def __init__(self):
        print("Base Constructor")

class Derived(Base):
    def __init__(self):  # Overriding Parent Constructor
        print("Derived Constructor")

# ✅ Method Signature Parsing (Matching Method by Signature)
class Example:
    def display(self, a: int, b: int) -> int:  # Method Signature
        return a + b

# ✅ Function Signature Parsing (Matching Function by Signature)
def function_signature(a: int, b: float) -> float:
    return a * b

# ✅ Constructor Signature Parsing (Matching Constructor by Signature)
class Vehicle:
    def __init__(self, model: str, year: int):
        self.model = model
        self.year = year

# Testing the Examples
print(precedence_example())  # Operator Precedence
print(arity_example(10))  # Operator Arity
print(show(10))  # Function Overloading
print(show("Python"))  # Function Overloading
print(Child().show())  # Function Overriding
print(Student("Alice", 22).name)  # Constructor Overloading
```

```
Derived()  # Constructor Overriding
math1 = Math(10)
math2 = Math(20)
print((math1 + math2).value)  # Operator Overloading
print(math1 > math2)  # Operator Overriding

# Note:
# 1️⃣ Operator Parsing → Overloading, Overriding, Precedence, Associativity, and
# 2️⃣ Function Parsing → Overloading (using @singledispatch) & Overriding (via su
# 3️⃣ Constructor Parsing → Overloading (handling multiple signatures) & Overrid
# 4️⃣ Method & Function Signature Parsing → Ensuring methods & functions match pa
```

```
Person(Name: Alice, Age: 25)
MagicMethodsDemo('Alice', 25)
55
5
True
25
l
A l i c e
True
Calling Alice with age 25
Entering context with Alice
Person(Name: Alice, Age: 25)
Exiting context with Alice
Attribute unknown not found!
{'class': <class '__main__.MagicMethodsDemo'>, 'bases': (<class 'object'>,), 'mr
o': (<class '__main__.MagicMethodsDemo'>, <class 'object'>), 'module': '__main_
_', 'dict': mappingproxy({'__module__': '__main__', '__slots__': ('name', 'age',
'index', 'new_attr'), '__init__': <function MagicMethodsDemo.__init__ at 0x000001
F2FE2D05E0>, '__str__': <function MagicMethodsDemo.__str__ at 0x000001F2FE2D0E00
>, '__repr__': <function MagicMethodsDemo.__repr__ at 0x000001F2FE2D0EA0>, '__add
__': <function MagicMethodsDemo.__add__ at 0x000001F2FE2D2F20>, '__sub__': <funct
ion MagicMethodsDemo.__sub__ at 0x000001F2FE2D2FC0>, '__mul__': <function MagicMe
thodsDemo.__mul__ at 0x000001F2FE2D2A20>, '__truediv__': <function MagicMethodsDe
mo.__truediv__ at 0x000001F2FE2D2AC0>, '__floordiv__': <function MagicMethodsDem
o.__floordiv__ at 0x000001F2FE2D3240>, '__mod__': <function MagicMethodsDemo.__mo
d__ at 0x000001F2FE2D32E0>, '__pow__': <function MagicMethodsDemo.__pow__ at 0x00
0001F2FE2D1E40>, '__eq__': <function MagicMethodsDemo.__eq__ at 0x000001F2FE2D1EE
0>, '__ne__': <function MagicMethodsDemo.__ne__ at 0x000001F2FE2D2020>, '__lt__':
<function MagicMethodsDemo.__lt__ at 0x000001F2FE2D20C0>, '__le__': <function Mag
icMethodsDemo.__le__ at 0x000001F2FE2D0CC0>, '__gt__': <function MagicMethodsDem
o.__gt__ at 0x000001F2FE2D0C20>, '__ge__': <function MagicMethodsDemo.__ge__ at 0
x000001F2FE2D0F40>, '__len__': <function MagicMethodsDemo.__len__ at 0x000001F2FE
2D0D60>, '__getitem__': <function MagicMethodsDemo.__getitem__ at 0x000001F2FE2D0
FE0>, '__setitem__': <function MagicMethodsDemo.__setitem__ at 0x000001F2FE2D1080
>, '__delitem__': <function MagicMethodsDemo.__delitem__ at 0x000001F2FE2D1120>,
'__iter__': <function MagicMethodsDemo.__iter__ at 0x000001F2FE2D11C0>, '__next_
_': <function MagicMethodsDemo.__next__ at 0x000001F2FE2D1260>, '__contains__': <
function MagicMethodsDemo.__contains__ at 0x000001F2FE2D1300>, '__call__': <funct
ion MagicMethodsDemo.__call__ at 0x000001F2FE2D13A0>, '__enter__': <function Magi
cMethodsDemo.__enter__ at 0x000001F2FE2D1440>, '__exit__': <function MagicMethods
Demo.__exit__ at 0x000001F2FE2D14E0>, '__getattr__': <function MagicMethodsDemo._
_getattr__ at 0x000001F2FE2D1580>, '__setattr__': <function MagicMethodsDemo.__se
tattr__ at 0x000001F2FE2D1620>, '__delattr__': <function MagicMethodsDemo.__delat
tr__ at 0x000001F2FE2D16C0>, 'get_class_info': <function MagicMethodsDemo.get_cla
ss_info at 0x000001F2FE2D1760>, '__hash__': <function MagicMethodsDemo.__hash__ a
t 0x000001F2FE2D1800>, '__bytes__': <function MagicMethodsDemo.__bytes__ at 0x000
001F2FE2D18A0>, '__format__': <function MagicMethodsDemo.__format__ at 0x000001F2
FE0D0040>, '__index__': <function MagicMethodsDemo.__index__ at 0x000001F2FE0D02C
0>, '__copy__': <function MagicMethodsDemo.__copy__ at 0x000001F2FE0D0360>, '__de
epcopy__': <function MagicMethodsDemo.__deepcopy__ at 0x000001F2FE0D0900>, '__siz
eof__': <function MagicMethodsDemo.__sizeof__ at 0x000001F2FE0D09A0>, 'age': <mem
ber 'age' of 'MagicMethodsDemo' objects>, 'index': <member 'index' of 'MagicMetho
dsDemo' objects>, 'name': <member 'name' of 'MagicMethodsDemo' objects>, 'new_att
r': <member 'new_attr' of 'MagicMethodsDemo' objects>, '__doc__': None}), 'name':
'MagicMethodsDemo', 'qualname': 'MagicMethodsDemo'}
1783039627156642475
b'Alice'
Formatted: Alice (25 years old)
48
New Point: (6, 8)
```

```
5
15
30
This is Child Class
Integer: 10
String: Hi
Default: 3.14
Bark
Unknown 18
Sahil 18
Kaushik 30
Child Constructor
True
15
6
B
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main
__.A'>, <class 'object'>]
14
3
-5 15 100
14
15
Integer: 10
String: Python
Child Class
Alice
Derived Constructor
30
False
```

8. Iterators, Generators, Coroutines, Closures & Decorators

In [ ]:

```python
## Iterators
# what is an iterator?
# An iterator is an object that allows you to traverse a container (like a list)
# what is the iter() function?
# The iter() function returns an iterator object for the given object.
# what is the next() function?
# The next() function returns the next item from the iterator.
# what is the StopIteration exception?
# The StopIteration exception is raised when there are no more items to return f
# what is the __iter__() method?
# The __iter__() method returns an iterator object for the given object.
# what is the __next__() method?
# The __next__() method returns the next item from the iterator.

# ✅ Creating an Iterator
class MyIterator:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self  # The iterator object itself

    def __next__(self):
        if self.current < self.end:
            self.current += 1
```

```
            return self.current - 1
        # else:
        #     raise StopIteration  # This should be avoided in Python 3.7+

# ✅ Creating an iterator object
iter_obj = MyIterator(3, 6)

print(next(iter_obj))  # Output: 3
print(next(iter_obj))  # Output: 4
print(next(iter_obj))  # Output: 5
print(next(iter_obj))  # Raises StopIteration (Expected Behavior)


# what is an iterable?
# An iterable is an object that can be iterated over.

# difference between iterator and iterable?
# An iterable is an object that can be iterated over, while an iterator is an ob

## Generators
# what is a generator?
# A generator is a function that returns an iterator object.
# what is the yield keyword?
# The yield keyword is used in a generator function to return a value without te
# what is the next() function?
# The next() function returns the next item from the generator.
# example of a generator function
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()

print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
# print(next(gen))  # Raises StopIteration

# what is a generator expression?
# A generator expression is a concise way to create a generator.
# example of a generator expression
gen = (x for x in range(1, 4))

print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
# print(next(gen))  # Raises StopIteration

# what is the send() method?
# The send() method is used to send a value to the generator.
# what is the close() method?
# The close() method is used to close the generator.
# what is the throw() method?
# The throw() method is used to raise an exception in the generator.

# difference between a generator and a list?
# A generator produces items one at a time, while a list produces all items at o
# difference between a generator and a function?
# A generator is a function that returns an iterator object, while a function re
```

```
# difference between a generator and an iterator?
# A generator is a function that returns an iterator object, while an iterator i

## Coroutines
# what is a coroutine?
# A coroutine is a function that can pause and resume its execution.
# what is the yield keyword?
# The yield keyword is used in a coroutine to pause its execution and return a v
# types of coroutines in Python?
# There are two types of coroutines in Python: generator-based coroutines and as
# what is the async keyword?
# The async keyword is used to define an async-based coroutine.
# what is the await keyword?
# The await keyword is used to pause the execution of an async-based coroutine u
# what is the asyncio module?
# The asyncio module provides support for writing asynchronous code using async-
# what is the async/await syntax?
# The async/await syntax is used to define and call async-based coroutines.
# example of a generator-based coroutine
def my_coroutine():
    while True:
        value = yield
        print(f"Received: {value}")

coro = my_coroutine()

next(coro)  # Start the coroutine
coro.send(10)  # Output: Received: 10
coro.send(20)  # Output: Received: 20

# example of an async-based coroutine
import asyncio

async def my_coroutine():
    print("Coroutine Started")
    await asyncio.sleep(1)  # Simulate async operation
    print("Coroutine Ended")

await my_coroutine()  # ✅ Works in Jupyter Notebook


# what is the aiohttp module?
# The aiohttp module is an asynchronous HTTP client/server framework based on as
# what is the asyncpg module?
# The asyncpg module is an asynchronous PostgreSQL client library based on async
# what is the asyncssh module?
# The asyncssh module is an asynchronous SSH client/server library based on asyn
# what is the asyncore module?
# The asyncore module provides support for asynchronous I/O handling.
# what is the asynchat module?
# The asynchat module provides support for asynchronous chat clients/servers.
# what is the selectors module?
# The selectors module provides high-level I/O multiplexing based on the select
# what is the trio module?
# The trio module is a friendly Python library for async concurrency and I/O.
# what is the curio module?
# The curio module is a library for concurrent I/O and async programming.
# what is the trio-asyncio module?
# The trio-asyncio module provides compatibility
# what is the async-timeout module?
```

```python
# The async-timeout module provides support for asynchronous timeouts.
# what is the asyncpgsa module?
# The asyncpgsa module is an asynchronous PostgreSQL client library for SQLAlche

## Itertools
# what is the itertools module?
# The itertools module provides a collection of tools for working with iterators
# what is the count() function?
# The count() function returns an infinite iterator that generates numbers start
# what is the cycle() function?
# The cycle() function returns an infinite iterator that cycles through a sequen
# what is the repeat() function?
# The repeat() function returns an iterator that repeats a value a specified num
# what is the accumulate() function?
# The accumulate() function returns an iterator that generates accumulated sums.
# what is the chain() function?
# The chain() function returns an iterator that chains together multiple iterabl
# what is the compress() function?
# The compress() function returns an iterator that filters elements from an iter
# what is the dropwhile() function?
# The dropwhile() function returns an iterator that drops elements from an itera
# what is the filterfalse() function?
# The filterfalse() function returns an iterator that filters elements from an i
# what is the groupby() function?
# The groupby() function returns an iterator that groups elements from an iterab
# what is the islice() function?
# The islice() function returns an iterator that slices elements from an iterabl
# what is the starmap() function?
# The starmap() function returns an iterator that applies a function to elements
# what is the takewhile() function?
# The takewhile() function returns an iterator that takes elements from an itera
# what is the tee() function?
# The tee() function returns multiple independent iterators from a single iterab
# what is the zip_longest() function?
# The zip_longest() function returns an iterator that aggregates elements from m
# what is the product() function?
# The product() function returns an iterator that generates the Cartesian produc
# what is the permutations() function?
# The permutations() function returns an iterator that generates all possible pe
# what is the combinations() function?
# The combinations() function returns an iterator that generates all possible co
# what is the combinations_with_replacement() function?
# The combinations_with_replacement() function returns an iterator that generate

import itertools

# ✅ count() Function
for i in itertools.count(1, 2):
    if i > 10:
        break
    print(i, end=" ")  # Output: 1 3 5 7 9

# ✅ cycle() Function
for i, j in zip(range(5), itertools.cycle("ABC")):
    print(i, j, end=" ")  # Output: 0 A 1 B 2 C 3 A 4 B

# ✅ repeat() Function
for i in itertools.repeat(10, 3):
    print(i, end=" ")  # Output: 10 10 10
```

```python
# ✅ accumulate() Function
data = [1, 2, 3, 4, 5]
result = list(itertools.accumulate(data))

print(result)  # Output: [1, 3, 6, 10, 15]

# ✅ chain() Function
data1 = [1, 2, 3]
data2 = [4, 5, 6]
result = list(itertools.chain(data1, data2))

print(result)  # Output: [1, 2, 3, 4, 5, 6]

# ✅ compress() Function
data = [1, 2, 3, 4, 5]
selectors = [True, False, True, False, True]
result = list(itertools.compress(data, selectors))

print(result)  # Output: [1, 3, 5]

# ✅ dropwhile() Function
data = [1, 2, 3, 4, 5]
result = list(itertools.dropwhile(lambda x: x < 3, data))

print(result)  # Output: [3, 4, 5]

# ✅ filterfalse() Function
data = [1, 2, 3, 4, 5]
result = list(itertools.filterfalse(lambda x: x < 3, data))

print(result)  # Output: [3, 4, 5]

# ✅ groupby() Function
data = [("A", 1), ("A", 2), ("B", 3), ("B", 4)]
result = {key: list(group) for key, group in itertools.groupby(data, key=lambda

print(result)  # Output: {'A': [('A', 1), ('A', 2)], 'B': [('B', 3), ('B', 4)]}

# ✅ islice() Function
data = [1, 2, 3, 4, 5]
result = list(itertools.islice(data, 2, None))

print(result)  # Output: [3, 4, 5]

# ✅ starmap() Function
data = [(1, 2), (3, 4), (5, 6)]
result = list(itertools.starmap(lambda x, y: x + y, data))

print(result)  # Output: [3, 7, 11]

# ✅ takewhile() Function
data = [1, 2, 3, 4, 5]
result = list(itertools.takewhile(lambda x: x < 3, data))

print(result)  # Output: [1, 2]

# ✅ tee() Function
data = [1, 2, 3, 4, 5]
iter1, iter2 = itertools.tee(data)
```

```
print(list(iter1))  # Output: [1, 2, 3, 4, 5]
print(list(iter2))  # Output: [1, 2, 3, 4, 5]

# ✅ zip_longest() Function
data1 = [1, 2, 3]
data2 = ["A", "B"]
result = list(itertools.zip_longest(data1, data2))

print(result)  # Output: [(1, 'A'), (2, 'B'), (3, None)]

# ✅ product() Function
data = [1, 2]
result = list(itertools.product(data, repeat=2))

print(result)  # Output: [(1, 1), (1, 2), (2, 1), (2, 2)]

# ✅ permutations() Function
data = [1, 2, 3]
result = list(itertools.permutations(data, 2))

print(result)  # Output: [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]

# ✅ combinations() Function
data = [1, 2, 3]
result = list(itertools.combinations(data, 2))

print(result)  # Output: [(1, 2), (1, 3), (2, 3)]

# ✅ combinations_with_replacement() Function
data = [1, 2, 3]
result = list(itertools.combinations_with_replacement(data, 2))

print(result)  # Output: [(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]

# Note:
# 1️⃣ count(), cycle(), repeat() → Infinite Iterators
# 2️⃣ accumulate(), chain(), compress() → Iterators
# 3️⃣ dropwhile(), filterfalse(), groupby() → Filtering Iterators
# 4️⃣ islice(), starmap(), takewhile() → Slicing & Mapping Iterators
# 5️⃣ tee(), zip_longest() → Multiple Iterators
# 6️⃣ product(), permutations(), combinations() → Combinatorial Iterators

## Closure
# what is a closure?
# A closure is a function that remembers the variables from its enclosing scope
# after the outer function has finished executing. This allows functions to reta
# state across multiple calls without using global variables.
# Closures are typically used when a function is returned from another function
# retains access to the variables of the outer function.

# types of closures in Python?
# There are two types of closures in Python: function closures and class closure
# what is a function closure?
# A function closure is a function that captures the environment in which it was
# what is a class closure?
# A class closure is a class that captures the environment in which it was defin

# example of a function closure
def outer_function(x):
    def inner_function(y): # Closure
```

```
        return x + y
    return inner_function

closure = outer_function(10)

print(closure(5))  # Output: 15

# example of a class closure
def outer_class(x):
    class InnerClass:
        def __init__(self, y):
            self.value = x + y
    return InnerClass

closure = outer_class(10)

print(closure(5).value)  # Output: 15

## Global Variables vs Closures
# 🔴 Problem: Global variables can be modified accidentally, leading to unexpecı
# ✔ No global variable needed! Each instance maintains its own state.
def make_counter():
    count = 0  # Encapsulated variable
    def increment():
        nonlocal count # instead of global count, we use nonlocal count - for en
        count += 1
        return count
    return increment  # Returning function

counter = make_counter()
print(counter())  # 1
print(counter())  # 2
print(counter())  # 3

# Real World Example:
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args}, {kwargs}")
        return func(*args, **kwargs)
    return wrapper  # Returning the inner function (closure)

@logger  # Applying the closure-based decorator
def add(a, b):
    return a + b

print(add(3, 5))  # Output: Calling add with (3, 5), {} → 8

# relationship between a closure and a decorator?
# A closure is a function that captures the environment in which it was defined,
```

```
3
4
5
None
1
2
3
1
2
3
Received: 10
Received: 20
Coroutine Started
Coroutine Ended
1 3 5 7 9 0 A 1 B 2 C 3 A 4 B 10 10 10 [1, 3, 6, 10, 15]
[1, 2, 3, 4, 5, 6]
[1, 3, 5]
[3, 4, 5]
[3, 4, 5]
{'A': [('A', 1), ('A', 2)], 'B': [('B', 3), ('B', 4)]}
[3, 4, 5]
[3, 7, 11]
[1, 2]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[(1, 'A'), (2, 'B'), (3, None)]
[(1, 1), (1, 2), (2, 1), (2, 2)]
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
[(1, 2), (1, 3), (2, 3)]
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
15
15
1
2
3
Calling add with (3, 5), {}
8
```

In [ ]:

```python
## Decorators
# A decorator in Python is a function that modifies the behavior of another func

#  🔹 How Decorators Work?
# 1️⃣ A decorator is a function that takes another function as input.
# 2️⃣ It wraps the original function with additional behavior.
# 3️⃣ It returns a new function with enhanced capabilities.

# Basic Decorator Example

def my_decorator(func):
    def wrapper():
        print("Before the function call")
        func()
        print("After the function call")
    return wrapper  # Returns the modified function

def say_hello():
    print("Hello, World!")

say_hello = my_decorator(say_hello)  # Manually applying the decorator
say_hello() # without decorator
```

```python
@my_decorator  # Applying the decorator
def say_hello():
    print("Hello, World!")

say_hello() # with decorator

## Decorators with Arguments
def my_decorator(func):
    def wrapper(*args, **kwargs):  # Accepts any number of arguments
        print(f"Calling {func.__name__} with arguments: {args}, {kwargs}")
        return func(*args, **kwargs)  # Calls the original function
    return wrapper

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

print(add(3, 5))

## Practical use case of decorators
# 1. Logging func calls
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned {result}")
        return result
    return wrapper

@log_decorator
def multiply(a, b):
    return a * b

multiply(4, 5)

# 2. Timing Execution
import time

def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time:.4f} seconds")
        return result
    return wrapper

@timer_decorator
def slow_function():
    time.sleep(2)
    print("Function Finished")

slow_function()

# 3. Enforcing Access Control
def require_admin(func):
    def wrapper(user):
        if user != "admin":
            print("Access Denied!")
```

```
            return
        return func(user)
    return wrapper

@require_admin
def delete_database(user):
    print(f"Database deleted by {user}")

delete_database("guest")  # Access Denied!
delete_database("admin")  # Database deleted

# 4. Caching with lru_cache
from functools import lru_cache

@lru_cache(maxsize=3)  # Caches results
def factorial(n):
    print(f"Calculating factorial({n})")
    return 1 if n == 0 else n * factorial(n - 1)

print(factorial(5))  # Computed
print(factorial(5))  # Cached

# 5. Preserving Metadata
import functools

def my_decorator(func):
    @functools.wraps(func)  # Preserves metadata
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper

## Types of Decorators in Python
# Python has different types of decorators based on their functionality and use
# 1️⃣ Function Decorators (Regular Decorators e.g. logging decorator)
# 2️⃣ Class Decorators
# 3️⃣ Method Decorators (Instance, Class, and Static Method Decorators)
# 4️⃣ Built-in Decorators

# 2. Class Decorator: Class decorators are used to modify class behavior instead
class CountInstances:
    def __init__(self, cls):
        self.cls = cls
        self.count = 0

    def __call__(self, *args, **kwargs):
        self.count += 1
        print(f"Creating instance {self.count} of {self.cls.__name__}")
        return self.cls(*args, **kwargs)

@CountInstances
class Person:
    def __init__(self, name):
        self.name = name

p1 = Person("Alice")
p2 = Person("Bob")

# 3. Method Decorators (Instance, Class & Static Methods)
# Python provides three built-in method decorators:
# 1️⃣ @staticmethod → For defining static methods
```

```python
# 2️⃣ @classmethod → For defining class methods
# 3️⃣ @property → For defining getter methods
class Example:
    class_var = "Class Variable"

    def __init__(self, value):
        self.value = value

    @staticmethod
    def static_method():
        print("This is a static method, does not access instance or class variab

    @classmethod
    def class_method(cls):
        print(f"This is a class method, it can access: {cls.class_var}")

    @property
    def value_squared(self):
        return self.value ** 2

# Using methods
ex = Example(5)
ex.static_method()  # Works without instance
ex.class_method()   # Can access class variables
print(ex.value_squared)  # Calls the property

# 4. Built-in Decorators in Python
# Python provides some predefined decorators for convenience:

# Decorator     Purpose
# @staticmethod: Defines a static method (no self or cls).
# @classmethod: Defines a class method (access cls).
# @property: Converts a method into a read-only attribute.
# @functools.lru_cache: Caches function results for faster execution.
# @functools.wraps: Preserves function metadata when using custom decorators.
# @dataclass: Automatically generates __init__, __repr__, etc.
# @singledispatch (for function overloading)

# 1️⃣ @staticmethod
# A static method does not receive the instance (self) or class (cls) as its fir
class MathOperations:
    @staticmethod
    def add(a, b):
        return a + b

print(MathOperations.add(5, 3))  # Output: 8
# When to use?
# When a method doesn't depend on the class or instance but is logically related

# 2️⃣ @classmethod
# A class method receives the class (cls) as its first parameter, allowing acces
class Employee:
    company = "TechCorp"

    @classmethod
    def change_company(cls, new_name):
        cls.company = new_name

Employee.change_company("InnovateTech")
print(Employee.company)  # Output: InnovateTech
```

```python
# When to use?
# When you need to modify class variables.
# When creating alternative constructors.

# 3️⃣ @property
# The @property decorator allows a method to be accessed like an attribute.
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def area(self):  # No need to call it like a method
        return 3.14 * self._radius ** 2

c = Circle(5)
print(c.area)  # Output: 78.5 (No parentheses)
# When to use?
# To define read-only properties.
# To encapsulate class attributes with getter/setter logic.

# 4️⃣ @functools.lru_cache (Least Recently Used Cache)
# This decorator caches function results to speed up execution.
from functools import lru_cache

@lru_cache(maxsize=3)  # Cache up to 3 results
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

print(fib(10))  # Output: 55 (Faster due to caching)
# When to use?
# When optimizing expensive recursive functions.
# When caching frequently used values.

# 5️⃣ @functools.wraps (Preserves Metadata)
# When creating custom decorators, @wraps ensures the wrapped function retains i
from functools import wraps

def my_decorator(func):
    @wraps(func)  # Preserves metadata
    def wrapper(*args, **kwargs):
        print("Before function call")
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def greet():
    """This function greets the user."""
    print("Hello!")

print(greet.__name__)  # Output: greet (without @wraps, it would return 'wrapper
print(greet.__doc__)   # Output: This function greets the user.
# When to use?
# When writing custom decorators to avoid losing function metadata.

# 6️⃣ @dataclass
# The @dataclass decorator automatically generates boilerplate code (e.g., __ini
from dataclasses import dataclass
```

```python
@dataclass
class Person:
    name: str
    age: int

p = Person("Alice", 30)
print(p)  # Output: Person(name='Alice', age=30)
# When to use?
# When creating data models without manually defining __init__, __repr__, etc.

# 7️⃣ @functools.singledispatch (Function Overloading)
# This decorator enables function overloading based on argument types.
from functools import singledispatch

@singledispatch
def process(data):
    raise NotImplementedError("Unsupported data type")

@process.register(int)
def _(data):
    return f"Processing integer: {data}"

@process.register(str)
def _(data):
    return f"Processing string: {data.upper()}"

print(process(10))   # Output: Processing integer: 10
print(process("abc"))  # Output: Processing string: ABC
# When to use?
# When implementing function overloading in Python.

## Abstraction & Decorator Relationship
# What is Abstraction in Python?
# Abstraction is an OOP (Object-Oriented Programming) principle that hides imple
# Key Idea: The user only needs to know what a function/class does, not how it d
# Implemented Using:
# 1. Abstract Classes (ABC module in Python)
# 2. Abstract Methods (Methods that must be implemented in child classes)
from abc import ABC, abstractmethod

class Animal(ABC):  # Abstract class
    @abstractmethod
    def make_sound(self):
        pass  # Must be implemented by subclasses

class Dog(Animal):
    def make_sound(self):
        return "Bark!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# animal = Animal()  # ❌ Error: Can't instantiate abstract class
dog = Dog()
cat = Cat()

print(dog.make_sound())  # ✅ Output: Bark!
print(cat.make_sound())  # ✅ Output: Meow!
# ✅ Here, Animal is an abstract class, and make_sound() must be implemented in
```

```python
## Relationship Between Abstraction & Decorators
# Python decorators can be used to enforce abstraction in multiple ways:
# 1️⃣ Using Decorators to Enforce Abstract Methods (@abstractmethod)
# Python provides @abstractmethod, a built-in decorator that ensures a method is
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod  # Decorator ensures implementation in subclasses
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

# shape = Shape()  # ❌ Error: Abstract class can't be instantiated
circle = Circle(5)  # ✅ Works
print(circle.area())  # Output: 78.5

# 2️⃣ Using Decorators to Implement Access Control (Simulating Private Methods)
# Python does not have truly private methods, but decorators can enforce restric
# Example: Protecting a Method Using a Decorator
def private_method_decorator(func):
    def wrapper(*args, **kwargs):
        raise RuntimeError("This method is private and cannot be accessed direct
    return wrapper

class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    @private_method_decorator
    def _calculate_interest(self):  # Simulating a private method
        return self.balance * 0.05

account = BankAccount(1000)
# account._calculate_interest()  # ❌ Raises an error
# ✅ Here, we used a decorator to restrict access to _calculate_interest() inste

# 3️⃣ Using Decorators to Control Method Behavior Dynamically
# Instead of using abstract methods, decorators can modify methods dynamically.
# Example: Adding Logging to Abstract Methods
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    @log_decorator  # Decorator on abstract method
    def move(self):
        pass
```

```python
class Car(Vehicle):
    def move(self):
        print("Car is moving!")

car = Car()
car.move()
# ✅ Here, the decorator @log_decorator applies logging functionality while enfo

## Chaining of decorators
# Chaining decorators means applying multiple decorators to a single function. E

# ✅ How Does It Work?
# When multiple decorators are applied to a function, they are executed from bot

# Execution Order: Decorators are applied bottom-up but executed top-down.
# Function Arguments: Decorators must handle arguments properly when chaining.
# Class Methods Chained: decorators can be applied to class methods.
# Built-in Decorators: Built-in decorators (@staticmethod, @classmethod) can be

# 1️⃣ Basic Example of Chaining Decorators
def uppercase_decorator(func):
    def wrapper():
        return func().upper()
    return wrapper

def exclamation_decorator(func):
    def wrapper():
        return func() + "!!!"
    return wrapper

@exclamation_decorator
@uppercase_decorator  # This runs first
def greet():
    return "hello"

print(greet())  # Output: HELLO!!!

# 📝 Execution Order:
# @uppercase_decorator runs first → Converts "hello" to "HELLO".
# @exclamation_decorator runs next → Appends "!!!", resulting in "HELLO!!!".
# ✅ The decorators are applied bottom-up, but execute top-down.

# 2️⃣ Chaining Decorators with Arguments
# Decorators can also handle function arguments while chaining.
def double_decorator(func):
    def wrapper(num):
        return func(num * 2)
    return wrapper

def square_decorator(func):
    def wrapper(num):
        return func(num) ** 2
    return wrapper

@square_decorator
@double_decorator
def add_five(num):
    return num + 5

print(add_five(3))  # Output: 64
```

```
# 📝 Execution Flow:
# double_decorator(3) → (3 * 2) = 6
# square_decorator(6 + 5) → (11 ** 2) = 121
# Final Output = 121
# ✅ Decorators modify input before passing it to the next layer!

# 3️⃣ Chaining Decorators on Class Methods
# Decorators can be applied to class methods, including @staticmethod and @class
def bold_decorator(func):
    def wrapper(*args, **kwargs):
        return f"<b>{func(*args, **kwargs)}</b>"
    return wrapper

def italic_decorator(func):
    def wrapper(*args, **kwargs):
        return f"<i>{func(*args, **kwargs)}</i>"
    return wrapper

class Formatter:
    @bold_decorator
    @italic_decorator
    def text(self, message):
        return message

f = Formatter()
print(f.text("Hello"))  # Output: <b><i>Hello</i></b>

# ✅ Class methods can have chained decorators just like functions!

# 4️⃣ Chaining Built-in Decorators
# Python supports built-in method decorators, and we can chain them with custom
import functools

def log_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

class Example:
    @staticmethod
    @log_decorator  # This runs first
    def greet():
        print("Hello from static method!")

Example.greet()

# 📝 Execution Order:

# @log_decorator prints "Calling function: greet".
# @staticmethod ensures greet() can be called without an instance.
# ✅ Python's built-in decorators can also be part of the decorator chain.
```

In [ ]: