Hey, this is **“Nagaraj Loni”**

Wel-Come to Python Programming Lang with OOP’s “ from *Basic to Advanced level Notes with Q&A”*

Topic:

1. Variables & Data Types

Variables: Storing a value e.g. age = 21, name = “Ns”.

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

Global Variables

```
def myfunc():
  global x
  x = "fantastic"
myfunc()
print("Python is " + x)     # Python is fantastic
```

Data types: int 21, str ‘ns’, “ns”, bool True, False, float 1.23, 1.3232.

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview
None Type:	NoneType

Interview Questions

1. What are variables in Python? How are they different from other languages?

Answer:

A **variable** in Python is a name that refers to a memory location where a value is stored. Unlike statically typed languages like C or Java, Python variables are **dynamically typed**, meaning you don't need to declare their type.

Example:

```
x = 10       # Integer
y = "Hello"  # String
z = 3.14     # Float


print(type(x))  # Output: <class 'int'>
print(type(y))  # Output: <class 'str'>
print(type(z))  # Output: <class 'float'>
```

---

2. What are Python's standard data types?

Answer:

Python provides the following **standard data types**:

- **Numeric Types:** int, float, complex
- **Sequence Types:** list, tuple, range
- **Text Type:** str
- **Set Types:** set, frozenset
- **Mapping Type:** dict
- **Boolean Type:** bool
- **Binary Types:** bytes, bytearray, memoryview

Example:

```
a = 10       # int
b = 10.5     # float
c = 2 + 3j   # complex
d = [1, 2]   # list
e = (3, 4)   # tuple
```

```
f = {5, 6}   # set
g = {'key': 'value'}  # dict
h = True     # bool

print(type(a), type(b), type(c), type(d), type(e), type(f), type(g), type(h))
```

---

3. What is the difference between Mutable and Immutable data types?

Answer:

- **Mutable** data types can be changed after creation (list, dict, set, bytearray).
- **Immutable** data types cannot be changed after creation (int, float, str, tuple, bool, frozenset).

Example:

```
# Mutable example
lst = [1, 2, 3]
lst.append(4)  # Changes the list
print(lst)  # Output: [1, 2, 3, 4]

# Immutable example
tup = (1, 2, 3)
# tup[0] = 10  # TypeError: 'tuple' object does not support item assignment
```

---

4. What is type casting in Python?

Answer:

Type casting (type conversion) is converting one data type into another.

Example:

```
x = "100"
y = int(x)  # String to Integer
z = float(x)  # String to Float

print(y, type(y))  # Output: 100 <class 'int'>
print(z, type(z))  # Output: 100.0 <class 'float'>
```

---

5. What is the difference between is and == in Python?

Answer:

- == compares **values** (content).
- is compares **object identity** (memory location).

Example:

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b)  # True (same values)
print(a is b)  # False (different memory locations)

c = a  # Assigning same reference
print(a is c)  # True (same memory location)
```

---

6. What is the difference between a list and a tuple?

Answer:

Feature	List (list)	Tuple (tuple)
Mutability	Mutable	Immutable
Performance	Slower	Faster
Memory Usage	More	Less
Syntax	[]	()

Example:

```
lst = [1, 2, 3]
tup = (1, 2, 3)

lst.append(4)  # Works
# tup.append(4)  # TypeError: 'tuple' object has no attribute 'append'
```

---

7. How does Python handle memory management for variables?

Answer:

Python uses **automatic memory management**, including:

- **Reference Counting**: Tracks the number of references to an object.
- **Garbage Collection**: Removes objects with zero references.
- **Heap Memory Allocation**: Stores objects dynamically.

Example:

```
import sys

a = [1, 2, 3]
b = a  # Both refer to the same object
print(sys.getrefcount(a))  # Output: 3 (1 for `b`, 1 for `a`, and 1 for `sys.getrefcount`)
```

---

8. What is f-strings in Python, and how are they different from other formatting methods?

Answer:

f-strings (formatted string literals) provide an efficient way to format strings.

Example:

```
name = "Alice"
age = 25

# Using f-strings
print(f"My name is {name} and I am {age} years old.")

# Using `.format()`
print("My name is {} and I am {} years old.".format(name, age))

# Using `%` operator
print("My name is %s and I am %d years old." % (name, age))
```

---

9. What is the difference between del and None in Python?

Answer:

- **del** removes a variable from memory.
- **Assigning None** removes the value but keeps the variable.

Example:

```
x = 10
del x
# print(x)  # NameError: name 'x' is not defined

y = 20
y = None  # Keeps variable `y` but with no value
print(y)  # Output: None
```

---

10. What is the difference between deep copy and shallow copy?

Answer:

- **Shallow Copy (copy.copy())**: Copies only references, not nested objects.
- **Deep Copy (copy.deepcopy())**: Copies everything, including nested objects.

Example:

```
import copy

# Original List
lst1 = [[1, 2], [3, 4]]

# Shallow Copy
lst2 = copy.copy(lst1)
lst2[0][0] = 100  # Changes both lst1 and lst2

# Deep Copy
lst3 = copy.deepcopy(lst1)
lst3[0][0] = 200  # Only changes lst3
```

```
print(lst1)  # Output: [[100, 2], [3, 4]]
print(lst2)  # Output: [[100, 2], [3, 4]]
print(lst3)  # Output: [[200, 2], [3, 4]]
```

2.Basic Operators, Conditions & Input

Input : name = input()

Print(name)

Operator	Description
()	Parentheses
**	Exponentiation
+x -x ~x	Unary plus, unary minus, and bitwise NOT
* / // %	Multiplication, division, floor division, and modulus
+ -	Addition and subtraction
<< >>	Bitwise left and right shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
== != > >= < <= is is not in not in	Comparisons, identity, and membership operators
not	Logical NOT

and	AND
or	OR

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3

//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3
:=	print(x := 3)	x = 3 print(x)

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4

not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
~	NOT	Inverts all the bits	~x
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

---

1. What are the different types of operators in Python?

Answer:

Python provides the following types of operators:

1. **Arithmetic Operators** (+, -, *, /, //, %, **)
2. **Comparison (Relational) Operators** (==, !=, >, <, >=, <=)
3. **Logical Operators** (and, or, not)
4. **Bitwise Operators** (&, |, ^, ~, <<, >>)
5. **Assignment Operators** (=, +=, -=, *=, /=, //=, %=, **=)
6. **Identity Operators** (is, is not)
7. **Membership Operators** (in, not in)

Example:

```
a = 10
b = 3
print(a + b)  # Arithmetic: 13
```

```
print(a > b)  # Comparison: True
print(a and b)  # Logical: 3 (truthy value)
print(a & b)  # Bitwise: 2 (1010 & 0011)
```

---

2. What is the difference between / and // operators in Python?

Answer:

- / **(Division Operator)** returns a floating-point division.
- // **(Floor Division Operator)** returns an integer by rounding down the result.

Example:

```
print(10 / 3)   # Output: 3.3333333333333335
print(10 // 3)  # Output: 3 (Rounded down)
```

---

3. How does the modulo operator % work in Python?

Answer:

The **modulo operator (%)** returns the remainder of a division.

Example:

```
print(10 % 3)  # Output: 1 (10 divided by 3 gives remainder 1)
print(-10 % 3)  # Output: 2 (Python ensures the result is always non-negative)
```

---

4. What is the exponentiation operator (**) in Python?

Answer:

The ** operator is used for raising a number to a power.

Example:

```
print(2 ** 3)  # Output: 8 (2^3)
print(5 ** 0.5)  # Output: 2.236 (Square root of 5)
```

---

5. What is the difference between is and == operators in Python?

Answer:

- == **compares values** (content of the objects).
- is **compares object identity** (memory location).

Example:

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a

print(a == b)  # True (same values)
print(a is b)  # False (different memory locations)
print(a is c)  # True (same memory location)
```

---

6. How does Python handle user input using input()?

Answer:

The input() function takes user input as a **string** by default. We can convert it into other data types.

Example:

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))  # Converts string input to integer

print(f" Hello, {name}! You are {age} years old.")
```

---

7. What happens if we use multiple inputs in a single input() call?

Answer:

We can use split() to take multiple inputs from a single line.

Example:

```
x, y = input("Enter two numbers: ").split()
print(f" You entered: {x} and {y}")
```

---

8. What is the difference between and, or, and not operators in Python?

Answer:

- and: Returns True if both conditions are True.
- or: Returns True if at least one condition is True.
- not: Negates a boolean value.

Example:

a = True

b = False

print(a and b) # False

print(a or b) # True

print(not a) # False

---

9. What are Bitwise Operators in Python?

Answer:

Bitwise operators work at the binary level.

Operator	Description
&	AND
`	`
^	XOR
~	NOT
<<	Left Shift
>>	Right Shift

Example:

a = 5 # 0101 in binary

b = 3 # 0011 in binary

print(a & b) # Output: 1 (0001)

print(a | b) # Output: 7 (0111)

```
print(a ^ b)  # Output: 6 (0110)   # Same [true-true / false-false] then 0.
print(~a)  # Output: -6 (inverted bits)
print(a << 1)  # Output: 10 (shifts left)
print(a >> 1)  # Output: 2 (shifts right)
```

---

10. What is the use of eval() in Python?

Answer:

The eval() function **executes an expression given as a string**.

Example:

```
expr = "3 + 4 * 2"
result = eval(expr)  # Evaluates "3 + 4 * 2" => 11
print(result)  # Output: 11
```

⚠ **Warning**: eval() can be dangerous if used with untrusted input.

3.if-elif-else

if a > b: print("a is greater than b")

---

1. What are conditional statements in Python?

Answer:

Conditional statements in Python are used to execute code blocks based on conditions. The main conditional statements are:

- if
- elif
- else

Example:

```
x = 10


if x > 0:
    print("Positive number")
elif x < 0:
    print("Negative number")
else:
    print("Zero")
```

Output:
Positive number

---

2. What is the difference between if-elif-else and multiple if statements?

Answer:

- if-elif-else: Only one block executes (once a condition is met, remaining checks are skipped).
- Multiple if statements: All conditions are checked, even if one is met.

Example:

```
x = 10


# Using if-elif-else
```

```
if x > 5:
    print("Greater than 5")
elif x > 8:
    print("Greater than 8")  # This won't execute
else:
    print("Something else")
```

Output:

Greater than 5

 # Using multiple ifs

```
if x > 5:
    print("Greater than 5")
if x > 8:
    print("Greater than 8")  # This will execute too
```

Output:

Greater than 5

Greater than 8

---

3. Can we use if statements inside another if? What is Nested if?

Answer:

Yes, nested if statements allow an if condition inside another if block.

Example:

```
x = 15

if x > 10:
    print("x is greater than 10")
    if x % 2 == 0:
        print("x is even")
    else:
        print("x is odd")
```

Output:

x is greater than 10

x is odd

---

4. What is the difference between if condition: and if (condition): in Python?

Answer:

Both are valid in Python. Unlike some languages like C, parentheses are not required in Python if statements.

Example:

```
x = 5


if x > 3:  # Correct way
    print("Valid")


if (x > 3):  # Also valid, but parentheses are unnecessary
    print("Also valid")
```

Output:

Valid

Also, valid

---

5. How does the pass statement work in conditional statements?

Answer:

The pass statement is a placeholder that does nothing. It is used when a condition is required but no action needs to be taken.

Example:

```
x = 5
if x > 0:
    pass  # Placeholder, avoids indentation error
else:
    print("Negative number")
```

Output: (No output, as pass does nothing)

```
Print("hello" == "HELLO")     # output: False
```

4. Chained Conditionals & Nested Statements

---

1. What are chained conditionals in Python?

Answer:

Chained conditionals use if, elif, and else to check multiple conditions sequentially. If one condition is True, the remaining checks are skipped.

Example:

```
x = 75


if x >= 90:
    print("Grade: A")
elif x >= 80:
    print("Grade: B")
elif x >= 70:
    print("Grade: C")
else:
    print("Grade: D")
```

Output:
Grade: C

---

2. How is a nested if statement different from a chained conditional?

Answer:

- Chained Conditionals (if-elif-else) check one condition at a time.
- Nested if Statements have if conditions inside another if block.

Example (Nested if):

```
x = 20


if x > 10:
    print("Greater than 10")
    if x > 15:
        print("Greater than 15")
```

```
    else:
        print("Between 10 and 15")
```

Output:

Greater than 10

Greater than 15

---

3. Can you use logical operators in chained conditionals?

Answer:

Yes! Logical operators (and, or, not) can be used in chained conditionals.

Example:

```
x = 25

if x > 10 and x < 30:
    print("x is between 10 and 30")
elif x == 30 or x == 40:
    print("x is 30 or 40")
else:
    print("x is something else")
```

Output:
x is between 10 and 30

---

4. What happens if we forget an else statement in a nested if?

Answer:

Nothing breaks, but the program may not handle all cases properly.

Example (No else in Nested if):

```
x = 12

if x > 10:
    print("Greater than 10")
    if x % 2 == 0:
        print("Even number")
```

Output:

Greater than 10

Even number

If x = 11, no output for odd numbers! To fix it, add an else inside the nested if. → Greater than 10

---

5. *Can if statements be written in a single line?*

Answer:

Yes! Python allows single-line if statements (Ternary Operators).

Example:

x = 10

print("Even") if x % 2 == 0 else print("Odd")

Output:
Even

---

5. For Loops

1. What is a for loop in Python?

Answer:

A for loop in Python is used to iterate over sequences such as lists, tuples, strings, sets, and dictionaries.

Example:

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

for fruit in fruits:
    print(fruit)
```

Output:

apple

banana

cherry

---

2. How do you iterate over a range of numbers using for loop?

Answer:

The range() function generates a sequence of numbers, which can be used in a for loop.

Example:

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

Output:

0

1

2

3

4

---

3. What are the different ways to use the range() function in a for loop?

Answer:

The range() function has three forms:

- 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 increments of step.

Example:

```
print("range(5):")
for i in range(5):
    print(i)

print("\nrange(2, 7):")
for i in range(2, 7):
    print(i)

print("\nrange(1, 10, 2):")
for i in range(1, 10, 2):
    print(i)
```

Output:

```
range(5):
0
1
2
3
4

range(2, 7):
2
3
4
5
6
```

range(1, 10, 2):

1

3

5

7

9

---

4. How do you iterate over a string using a for loop?

Answer:

Since a string is a sequence of characters, a for loop can iterate over it.

Example:

```
word = "Python"
for char in word:
    print(char)
```

Output:

P

y

t

h

o

n

---

5. What is the difference between for and while loops in Python?

Answer:

- A for loop iterates over sequences (lists, strings, tuples, etc.).
- A while loop runs as long as a condition is True.

Example (for loop - used for definite iteration):

```
for i in range(3):
    print("Hello")
```

Output:

Hello

Hello

Hello

Example (while loop - used for indefinite iteration):

```
i = 0
while i < 3:
    print("Hello")
    i += 1
```

Output:

Hello

Hello

Hello

---

6. How do you use the break statement in a for loop?

Answer:

The break statement exits the loop immediately when a condition is met.

Example:

```
for num in range(10):
    if num == 5:
        break
    print(num)
```

Output:

0

1

2

3

4

---

7. How do you use the continue statement in a for loop?

Answer:

The continue statement skips the current iteration and moves to the next.

Example:

```
for num in range(5):
    if num == 2:
        continue
    print(num)
```

Output:

0

1

3

4

(Note: 2 is skipped.)

---

8. How do you use the else clause in a for loop?

Answer:

The else block executes only if the loop completes normally (without break).

Example:

```
for num in range(3):
    print(num)
else:
    print("Loop finished successfully")
```

Output:

0

1

2

Loop finished successfully

Example (with break - else won't execute):

```
for num in range(3):
    if num == 1:
        break
```

```
    print(num)
else:
    print("Loop finished successfully")
```

Output:

0

(Note: The loop exits early, so else doesn't run.)

---

9. How do you iterate over a dictionary using a for loop?

Answer:

Dictionaries have keys and values, and you can iterate over them using .keys(), .values(), or .items().

Example:

```
student = {"name": "Alice", "age": 20, "grade": "A"}


print("Iterate over keys:")
for key in student:
    print(key)


print("\nIterate over values:")
for value in student.values():
    print(value)


print("\nIterate over key-value pairs:")
for key, value in student.items():
    print(f"{key}: {value}")
```

Output:

Iterate over keys:

name

age

grade

Iterate over values:

Alice

20

A

Iterate over key-value pairs:

name: Alice

age: 20

grade: A

---

10. How do you use a nested for loop in Python?

Answer:

A nested for loop is a loop inside another loop. It is often used for working with matrices or generating patterns.

Example:

```
for i in range(3):
    for j in range(2):
        print(f" i ={i}, j={j}")
```

Output:

i=0, j=0

i=0, j=1

i=1, j=0

i=1, j=1

i=2, j=0

i=2, j=1

---

6. While Loop

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")

# output:
1
2
3
4
5
i is no longer less than 6
```

---

1. What is a while loop in Python?

Answer:

A while loop executes a block of code **as long as** a given condition is True.

Example:

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

Output:

1

2

3

4

5

(The loop stops when x becomes 6.)

---

2. What is the difference between a for loop and a while loop?

Answer:

- **for loop**: Used for **iterating over a sequence** (e.g., lists, tuples, strings, etc.).
- **while loop**: Used for **executing a block of code until a condition becomes False**.

Example (for loop - definite iteration):

```
for i in range(5):
    print(i)
```

Output:

0

1

2

3

4

Example (while loop - indefinite iteration):

```
x = 0
while x < 5:
    print(x)
    x += 1
```

Output:

0

1

2

3

4

---

3. What happens if the condition in a while loop is always True?

Answer:

This results in an **infinite loop**, which never stops unless you use break or manually interrupt it.

Example (Infinite Loop):

```
while True:
    print("This is an infinite loop")  # This will run forever
```

Solution: Use break to stop it.

```
x = 0
while True:
    print(x)
    x += 1
    if x == 5:
        break  # Stops the loop
```

Output:

```
0
1
2
3
4
```

---

4. How do you use the break statement in a while loop?

Answer:

The break statement **exits** the loop immediately when encountered.

Example:

```
x = 1
while x < 10:
    print(x)
    if x == 5:
        break  # Exit loop when x is 5
    x += 1
```

Output:

1

2

3

4

5

(The loop stops when x == 5.)

---

5. How do you use the continue statement in a while loop?

Answer:

The continue statement **skips** the current iteration and moves to the next.

Example:

```
x = 0
while x < 5:
    x += 1
    if x == 3:
        continue  # Skip the iteration when x is 3
    print(x)
```

Output:

1

2

4

5

(3 is skipped.)

---

6. How does the else clause work with a while loop?

Answer:

The else block executes **only if** the while loop **terminates normally** (without break).

Example:

```
x = 0
```

```
while x < 3:
    print(x)
    x += 1
else:
    print("Loop finished successfully")
```

Output:

0

1

2

Loop finished successfully

Example (with break - else won't execute):

```
x = 0
while x < 3:
    print(x)
    if x == 1:
        break
    x += 1
else:
    print("Loop finished successfully")
```

Output:

0

1

(The loop exits early, so else doesn't run.)

---

7. How do you create a countdown timer using a while loop?

Answer:

A while loop can be used to create a countdown.

Example:

```
import time  # Import time module


x = 5
while x > 0:
    print(x)
    time.sleep(1)  # Wait for 1 second
    x -= 1

print("Time's up!")
```

Output (with 1-second pauses between numbers):

5

4

3

2

1

Time's up!

---

8. How do you use a while loop to iterate over a list?

Answer:

Use an **index variable** to iterate through a list in a while loop.

Example:

```
fruits = ["apple", "banana", "cherry"]
i = 0


while i < len(fruits):
    print(fruits[i])
```

```
    i += 1
```

Output:

apple

banana

cherry

---

9. How can you implement a user input loop using while?

Answer:

A while loop is useful when taking user input until a condition is met.

Example:

```
while True:
    user_input = input("Enter 'exit' to stop: ")
    if user_input.lower() == "exit":
        print("Exiting...")
        break
    print(f" You entered: {user_input}")
```

Output (User interaction example):

Enter 'exit' to stop: hello

You entered: hello

Enter 'exit' to stop: python

You entered: python

Enter 'exit' to stop: exit

Exiting...

(The loop runs until the user types "exit".)

---

10. How do you use a nested while loop in Python?

Answer:

A **nested while loop** is a loop inside another while loop.

Example (Printing a pattern):

```
i = 1
```

```
while i <= 3:
    j = 1
    while j <= 2:
        print(f" i={i}, j={j}")
        j += 1
    i += 1
```

Output:

i=1, j=1

i=1, j=2

i=2, j=1

i=2, j=2

i=3, j=1

i=3, j=2

(Inner loop runs completely for each iteration of the outer loop.)

7. List [ ]

List items are ordered, **changeable**, and **allow duplicate values**.

List items are indexed, the first item has index [0], the second item has index [1] etc.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	**Add** the elements of a list (or any iterable), **to the end** of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the **specified position**
pop()	**Removes** the element at the specified position
remove()	Removes the item with the **specified value**
reverse()	Reverses the order of the list
sort()	Sorts the list

1. What is a list in Python? How do you create one?

Answer:

A **list** in Python is a **mutable, ordered** collection that can store elements of different data types.

Example:

```
# Creating a list
my_list = [10, "Python", 3.14, True]
print(my_list)
```

Output:

```
[10, 'Python', 3.14, True]
```

---

2. What are the different ways to access elements in a list?

Answer:

Elements in a list can be accessed using:

- **Indexing**
- **Negative Indexing**
- **Slicing**

Example:

```
my_list = [10, 20, 30, 40, 50]

print(my_list[0])    # First element
print(my_list[-1])   # Last element (negative indexing)
print(my_list[1:4])  # Slicing (from index 1 to 3)
```

Output:

```
10
50
[20, 30, 40]
```

---

3. How do you modify an element in a list?

Answer:

Lists are **mutable**, so you can modify elements using **indexing**.

Example:

my_list = [1, 2, 3, 4, 5]

my_list[2] = 99 # Changing the third element

print(my_list)

Output:

[1, 2, **99,** 4, 5]

---

4. How do you add elements to a list?

Answer:

Elements can be added using:

- append(): Adds a **single** element at the **end**.
- extend(): Adds **multiple** elements at the **end**.
- **insert(): Adds an element at a specific index.**

Example:

my_list = [1, 2, 3]

my_list.append(4) # Adds 4 at the end

my_list.extend([5, 6]) # Adds multiple elements

my_list.insert(1, 99) # Inserts 99 at index 1

print(my_list)

Output:

[1, 99, 2, 3, 4, 5, 6]

---

5. How do you remove elements from a list?

Answer:

Elements can be removed using:

- remove(value): Removes first occurrence of a value.
- pop(index): Removes and returns an element at a given index.
- del: Deletes an element by index.
- clear(): Removes all elements.

Example:

```
my_list = [10, 20, 30, 40, 50]


my_list.remove(30)   # Removes 30
my_list.pop(1)       # Removes element at index 1
del my_list[0]       # Deletes first element
# my_list.clear()    # Uncomment to remove all elements


print(my_list)
```

Output:

```
[40, 50]
```

---

6. How do you iterate through a list?

Answer:

You can use:

- **for loop**
- **while loop**
- **List comprehension**

Example:

```
my_list = ["Python", "Java", "C++"]


print("Using for loop:")
for item in my_list:
```

```
    print(item)

print("\nUsing while loop:")
i = 0
while i < len(my_list):
    print(my_list[i])
    i += 1

print("\nUsing list comprehension:")
[print(item) for item in my_list]
```

Output:

```
Using for loop:
Python
Java
C++

Using while loop:
Python
Java
C++

Using list comprehension:
Python
Java
C++
```

---

7. How do you check if an element exists in a list?

Answer:

Use the in keyword to check for existence.

Example:

```
my_list = [10, 20, 30, 40]

print(20 in my_list)   # True
print(50 in my_list)   # False
```

Output:

True

False

---

8. How do you sort a list in Python?

Answer:

Use:

- sort(): Sorts the list **in-place**.
- sorted(): Returns a **new sorted list**.

Example:

```
numbers = [5, 2, 9, 1, 5]

numbers.sort()   # Sort in ascending order
print(numbers)

numbers.sort(reverse=True)  # Sort in descending order
print(numbers)

new_sorted_list = sorted(numbers)  # Returns a sorted copy
print(new_sorted_list)
```

Output:

[1, 2, 5, 5, 9]

[9, 5, 5, 2, 1]

[1, 2, 5, 5, 9]

---

9. How do you find the length, minimum, and maximum values in a list?

Answer:

Use:

- len(): Finds length of the list.
- min(): Finds the smallest value.
- max(): Finds the largest value.

Example:

numbers = [10, 5, 30, 25]

print("Length:", len(numbers))

print("Minimum:", min(numbers))

print("Maximum:", max(numbers))

Output:

Length: 4

Minimum: 5

Maximum: 30

---

10. What is list comprehension? How does it work?

Answer:

List comprehension is a **shorter and more efficient way** to create lists.

Example (Creating a list of squares):

squares = [x**2 for x in range(1, 6)]

print(squares)

Output:

[1, 4, 9, 16, 25]

Example (Filtering even numbers):

numbers = [1, 2, 3, 4, 5, 6]

```
evens = [x for x in numbers if x % 2 == 0]
print(evens)
```

Output:

```
[2, 4, 6]
```

---

8. Tuples ( ) :

- Tuples are used to store multiple items in a single variable.
- Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
- A tuple is a collection which is ordered and **unchangeable**.
- Tuples are written with round brackets.
- Tuple items are ordered, unchangeable, and **allow duplicate values**.
- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

1. What is a tuple in Python? How is it different from a list?

Answer:

A tuple is an ordered, immutable collection of elements, meaning its values cannot be changed after creation.
Differences between List and Tuple:

Feature	List (list)	Tuple (tuple)
Mutable?	Yes (Can be modified)	No (Immutable)
Performance	Slower (More memory)	Faster (Less memory)
Syntax	[] (Square brackets)	() (Parentheses)

Example:

```
# Creating a tuple
my_tuple = (10, "Python", 3.14, True)
print(my_tuple)
```

Output:

```
(10, 'Python', 3.14, True)
```

---

2. How do you access elements in a tuple?

Answer:

Elements in a tuple can be accessed using indexing, negative indexing, and slicing.

Example:

```
my_tuple = (10, 20, 30, 40, 50)

print(my_tuple[0])   # First element
print(my_tuple[-1])  # Last element
print(my_tuple[1:4]) # Slicing from index 1 to 3
```

Output:

```
10
50
(20, 30, 40)
```

---

3. Can we modify a tuple in Python? Why or why not?

Answer:

No, tuples are immutable, meaning elements cannot **be changed, added, or removed after creation.**

Example (Trying to modify a tuple - This will cause an error):

```
my_tuple = (1, 2, 3)
# my_tuple[0] = 100  # TypeError: 'tuple' object does not support item assignment
```

---

4. How can you create a tuple with a single element?

Answer:

A tuple with one element must have a trailing comma; otherwise, it is considered an integer or string.

Example:

```
single_element_tuple = (5,)  # Correct
not_a_tuple = (5)  # This is just an integer

print(type(single_element_tuple))  # <class 'tuple'>
print(type(not_a_tuple))  # <class 'int'>
```

Output:

<class 'tuple'>

<class 'int'>

---

5. How can you add or remove elements from a tuple?

Answer:

Since tuples are immutable, you cannot directly add or remove elements. Instead, you need to convert the tuple to a list, modify it, and then convert it back to a tuple.

Example:

```
my_tuple = (1, 2, 3)
temp_list = list(my_tuple)  # Convert to list

temp_list.append(4)  # Add an element
temp_list.remove(2)  # Remove an element

new_tuple = tuple(temp_list)  # Convert back to tuple
print(new_tuple)
```

Output:

(1, 3, 4)

---

6. How can you unpack elements from a tuple?

Answer:

Tuple unpacking allows assigning tuple values to variables in a single line.

Example:

```
my_tuple = (10, 20, 30)

a, b, c = my_tuple  # Unpacking
print(a, b, c)
```

Output:

10 20 30

---

7. How can you iterate through a tuple?

Answer:

You can iterate over a tuple using a for loop.

Example:

```
my_tuple = ("Python", "Java", "C++")

for item in my_tuple:
    print(item)
```

Output:

Python

Java

C++

---

8. What are some common built-in tuple methods?

Answer:

Tuples have limited built-in methods compared to lists. The most commonly used ones are:

- count(value): Counts occurrences of a value.
- index(value): Returns the index of a value.

Example:

```
my_tuple = (1, 2, 3, 4, 2, 5)

print(my_tuple.count(2))  # Output: 2 (because 2 appears twice)
print(my_tuple.index(3))  # Output: 2 (index of 3)
```

Output:

2

2

---

9. How do you merge two or more tuples?

Answer:

Tuples can be merged using the + operator.

Example:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

merged_tuple = tuple1 + tuple2
print(merged_tuple)
```

Output:

(1, 2, 3, 4, 5, 6)

---

10. What is tuple comprehension? Can we use list comprehension with tuples?

Answer:

Tuple comprehension does not exist in Python. However, you can use generator expressions inside parentheses.

Example:

```
# This is NOT tuple comprehension but a generator expression
gen = (x**2 for x in range(5))
print(tuple(gen))  # Converting generator to tuple
```

Output:

(0, 1, 4, 9, 16)

(Using () creates a generator, not a tuple. To get a tuple, use tuple(gen).)

Set { }

* Sets are used to store multiple items in a single variable.

* Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

* A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.

Unordered

Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Unchangeable

Set items are unchangeable, meaning that we cannot change the items after the set has been created.

1. What is a set in Python? How is it different from a list or tuple?

Answer:

A set is an unordered, mutable, and unindexed collection of unique elements in Python.
Key Differences:

Feature	List (list)	Tuple (tuple)	Set (set)
Mutable?	Yes	No	Yes
Duplicates?	Allowed	Allowed	**Not Allowed**
Ordered?	Yes	Yes	No (Unordered)

Example:

```
my_set = {1, 2, 3, 4, 4, 5}  # Duplicate '4' is automatically removed
print(my_set)
```

Output:

{1, 2, 3, 4, 5}

---

2. How do you create a set in Python?

Answer:

A set can be created using curly braces {} or the set() constructor.

Example:

```
# Using curly braces
set1 = {10, 20, 30}

# Using set() constructor
set2 = set([40, 50, 60])  # Converting list to set

print(set1, set2)
```

Output:

{10, 20, 30} {40, 50, 60}

---

3. How do you add elements to a set?

Answer:

Use add() to add a single element and update() to add multiple elements.

Example:

```
my_set = {1, 2, 3}

my_set.add(4)        # Adds single element
my_set.update([5, 6]) # Adds multiple elements

print(my_set)
```

Output:

{1, 2, 3, 4, 5, 6}

---

4. How do you remove elements from a set?

Answer:

You can use:

- remove(value): Removes a specific element (throws an error if not found).
- discard(value): Removes an element (no error if not found).
- pop(): Removes and returns a random element.
- clear(): Removes all elements.

Example:

my_set = {10, 20, 30, 40}

my_set.remove(20) # Removes 20 (error if not found)

*my_set.**discard**(50) # No error if 50 is not found*

removed_element = my_set.pop() # Removes a random element

my_set.clear() # Removes all elements

print(my_set)

Output:

set() # Empty set

---

5. How do you check if an element exists in a set?

Answer:

Use the in keyword.

Example:

my_set = {1, 2, 3, 4, 5}

print(3 in my_set) # True

print(10 in my_set) # False

Output:

True

False

---

6. What are set operations (union, intersection, difference, symmetric difference) in Python?

Answer:

Python provides built-in methods for set operations:

- **union**(): Combines two sets (| operator).
- **intersection**(): Common elements (& operator).
- **difference**(): Elements in first set but not in second (- operator).
- **symmetric_difference**(): Elements in either set but not both (^ operator).

Example:

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

print("Union:", A | B)                  # {1, 2, 3, 4, 5, 6}
print("Intersection:", A & B)           # {3, 4}
print("Difference (A - B):", A - B)     # {1, 2}
print("Symmetric Difference:", A ^ B)   # {1, 2, 5, 6}
```

---

7. How do you copy a set in Python?

Answer:

You can copy a set using:

- copy() method (shallow copy).
- set() constructor.

Example:

```
original_set = {1, 2, 3}
copy_set = original_set.copy()  # Using copy method
copy_set2 = set(original_set)   # Using set() constructor

print(copy_set, copy_set2)
```

Output:

{1, 2, 3} {1, 2, 3}

---

8. How do you check if one set is a subset, superset, or disjoint set of another?

Answer:

- issubset(): Checks if all elements of one set are in another.
- issuperset(): Checks if a set contains all elements of another set.
- isdisjoint(): Checks if two sets have no common elements.

Example:

A = {1, 2, 3}

B = {1, 2, 3, 4, 5}

C = {6, 7, 8}

print(A.issubset(B)) # True

print(B.issuperset(A)) # True

print(A.isdisjoint(C)) # True (No common elements)

Output:

True

True

True

---

9. How do you convert a list or tuple to a set?

Answer:

Use the set() constructor.

Example:

 # Converting list to set

my_list = [1, 2, 2, 3, 4]

set_from_list **= set(my_list)**

Converting tuple to set

```
my_tuple = (5, 6, 6, 7)
set_from_tuple = set(my_tuple)

print(set_from_list, set_from_tuple)
```

Output:

```
{1, 2, 3, 4} {5, 6, 7}
```

---

10. What is a frozen set in Python? How is it different from a normal set?

Answer:

A frozen set is an immutable version of a set. It cannot be modified (no add/remove).

Example:

```
my_set = {1, 2, 3}
frozen = frozenset(my_set)

# frozen.add(4)  # AttributeError: 'frozenset' object has no attribute 'add'
print(frozen)
```

Output:

```
frozenset({1, 2, 3})
```

9. Dictionary

- Dictionaries are used to store data values in key: value pairs.
- A dictionary is a collection which is ordered*, changeable and do not **allow duplicates.**

Dictionary Items

- Dictionary items are **ordered**, **changeable**, and do **not allow duplicates**.
- Dictionary items are presented in key: value pairs, and can be referred to by using the key name.

- When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.

- Unordered means that the items do not have a defined order, you cannot refer to an item by using an index.

** *Changeable*

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

** *Duplicates Not Allowed*

Dictionaries cannot have two items with the same key:

---

1. What is a dictionary in Python? How is it different from a list?

Answer:

A dictionary in Python is an unordered, mutable, key-value pair collection. It is optimized for fast lookups.

Feature	List (list)	Dictionary (dict)
Data Structure	Ordered collection of elements	Unordered collection of key-value pairs
Access	By index (list[0])	By key (dict['key'])
Mutable?	Yes	Yes

Feature	List (list)	Dictionary (dict)
Duplicates?	Allowed	Keys must be unique

Example:

my_dict = {"name": "Alice", "age": 25, "city": "New York"}

print(my_dict)

Output:

{'name': 'Alice', 'age': 25, 'city': 'New York'}

---

2. How do you create a dictionary in Python?

Answer:

You can create a dictionary using **curly braces {} or the dict()** constructor.

Example:

Using curly braces

dict1 = {"name": "John", "age": 30}

Using dict() constructor

dict2 = dict(name="Alice", age=25)

print(dict1, dict2)

Output:

{'name': 'John', 'age': 30} {'name': 'Alice', 'age': 25}

---

3. How do you access values in a dictionary?

Answer:

Use keys with [] or .get() method.

Example:

my_dict = {"name": "Bob", "age": 27}

print(my_dict["name"]) # Access using key

print(my_dict.get("age")) # Access using get()

Output:

Bob

27

---

4. How do you add, update, and delete key-value pairs in a dictionary?

Answer:

- Add: Assign a value to a new key.
- Update: Use update() or direct assignment.
- Delete: Use del, pop(), or popitem().

Example:

```
my_dict = {"name": "Eve", "age": 28}

# Adding a new key-value pair
my_dict["city"] = "London"

# Updating a value
my_dict["age"] = 29

# Deleting a key-value pair
del my_dict["city"]
# my_dict.pop("name")  # Alternative

print(my_dict)
```

Output:

{'name': 'Eve', 'age': 29}

---

5. What are dictionary keys and values? Can a dictionary have duplicate keys?

Answer:

- **Keys must be unique** and immutable (strings, numbers, tuples).
- Values can be mutable and duplicated.

Example:

```
my_dict = {
   "name": "Alice",
   "age": 25,
   "age": 30  # Duplicate key (overwrites previous value)
}
print(my_dict)  # 'age' will have the latest value
```

Output:

```
{'name': 'Alice', 'age': 30}
```

---

6. How do you iterate over a dictionary?

Answer:

Use a for loop with .items(), .keys(), or .values().

Example:

```
my_dict = {"name": "David", "age": 32, "city": "Paris"}

# Iterate through keys and values
for key, value in my_dict.items():
   print(f"{key}: {value}")
```

Output:

```
name: David
age: 32
city: Paris
```

---

7. What are the common dictionary methods?

Answer:

Some important dictionary methods are:

- **keys**() → Returns all keys.
- **values**() → Returns all values.
- **items**() → Returns key-value pairs.
- **get**(key, default) → Retrieves value (avoids KeyError).
- **pop**(key) → Removes key and returns value.
- **update**(dict2) → Merges dictionaries.

Example:

```
my_dict = {"a": 1, "b": 2, "c": 3}

print(my_dict.keys())    # dict_keys(['a', 'b', 'c'])
print(my_dict.values())  # dict_values([1, 2, 3])
print(my_dict.items())   # dict_items([('a', 1), ('b', 2), ('c', 3)])

print(my_dict.get("b", "Not Found"))  # 2
print(my_dict.get("x", "Not Found"))  # Not Found
```

---

8. How do you merge two dictionaries?

Answer:

Use update(), {**dict1, **dict2}, or the | operator (Python 3.9+).

Example:

```
dict1 = {"x": 10, "y": 20}
dict2 = {"y": 25, "z": 30}

# Using update()
merged_dict1 = dict1.copy()
merged_dict1.update(dict2)
```

```
# Using dictionary unpacking (Python 3.5+)
merged_dict2 = {**dict1, **dict2}

# Using '|' operator (Python 3.9+)
merged_dict3 = dict1 | dict2

print(merged_dict1, merged_dict2, merged_dict3)
```

Output:

{'x': 10, 'y': 25, 'z': 30} {'x': 10, 'y': 25, 'z': 30} {'x': 10, 'y': 25, 'z': 30}

---

9. What is a dictionary comprehension?

Answer:

Dictionary comprehension is a concise way to create dictionaries.

Example:

```
squares = {x: x**2 for x in range(5)}
print(squares)
```

Output:

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

---

10. What is the difference between deepcopy() and copy() for dictionaries?

Answer:

- copy() → Creates a shallow copy (modifies nested objects in both).
- deepcopy() → Creates a deep copy (modifies only the copied dictionary).

Example:

```
import copy

original = {"a": 1, "b": [10, 20]}
shallow_copy = original.copy()
deep_copy = copy.deepcopy(original)
```

```
shallow_copy["b"].append(30)  # Affects both original and shallow copy

print(original)      # {'a': 1, 'b': [10, 20, 30]}
print(deep_copy)     # {'a': 1, 'b': [10, 20]}  # Unaffected
```

String Methods:

capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isascii()	Returns True if all characters in the string are ascii characters

isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Converts the elements of an iterable into a string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string

rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Top questions on **String Methods**

1. What are the different ways to create a string in Python?
   Explain single, double, and triple quotes, along with raw strings (r'') and formatted strings (f'').
2. What is the difference between find() and index() methods in Python?
   Both are used to locate substrings, but find() returns -1 if not found, whereas index() raises an error.
3. How do split() and join() methods work in Python?
   Explain how split() breaks a string into a list, while join() combines a list into a string.

4. What does the strip() method do? How is it different from lstrip() and rstrip()?
   Removes whitespace (or specified characters) from both ends, while lstrip() removes from the left and rstrip() from the right.
5. How does the replace() method work in Python? Can it be used for multiple replacements?
   Replaces a substring with another, and yes, chaining or using dictionaries can help with multiple replacements.
6. What is the difference between startswith() and endswith() methods?
   Checks if a string starts or ends with a specific substring and returns True or False.
7. How do isalpha(), isdigit(), isalnum(), and isspace() methods work?
   Checks if the string contains only letters, numbers, both, or only spaces, respectively.
8. What is the difference between lower(), upper(), title(), and capitalize() methods?
   Converts to lowercase, uppercase, title case, and capitalizes the first letter of the string, respectively.
9. How can you check if a string is a palindrome using Python string methods?
10. def is_palindrome(s):
11. return s == s[::-1]
12. print(is_palindrome("madam")) # True
13. How do you count occurrences of a substring in a string?
    Use the count() method:
14. text = "hello world, hello Python"
15. print(text.count("hello")) # Output: 2

Slice Operator

The slice() function returns a slice object.

A slice object is used to specify how to slice a sequence. You can specify where to start the slicing, and where to end. You can also specify the step, which allows you to e.g. slice only every other item.

Syntax

slice(*start*, *end, step*)

Function	Description
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number
delattr()	Deletes the specified attribute (property or method) from the specified object
dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided by argument2
enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object

eval()	Evaluates and executes an expression
exec()	Executes the specified code (or object)
filter()	Use a filter function to exclude items in an iterable object
float()	Returns a floating point number
format()	Formats a specified value
frozenset()	Returns a frozenset object
getattr()	Returns the value of the specified attribute (property or method)
globals()	Returns the current global symbol table as a dictionary
hasattr()	Returns True if the specified object has the specified attribute (property/method)
hash()	Returns the hash value of a specified object
help()	Executes the built-in help system
hex()	Converts a number into a hexadecimal value
id()	Returns the id of an object
input()	Allowing user input
int()	Returns an integer number
isinstance()	Returns True if a specified object is an instance of a specified object
issubclass()	Returns True if a specified class is a subclass of a specified object
iter()	Returns an iterator object
len()	Returns the length of an object

list()	Returns a list
locals()	Returns an updated dictionary of the current local symbol table
map()	Returns the specified iterator with the specified function applied to each item
max()	Returns the largest item in an iterable
memoryview()	Returns a memory view object
min()	Returns the smallest item in an iterable
next()	Returns the next item in an iterable
object()	Returns a new object
oct()	Converts a number into an octal
open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
print()	Prints to the standard output device
property()	Gets, sets, deletes a property
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
repr()	Returns a readable version of an object
reversed()	Returns a reversed iterator
round()	Rounds a numbers
set()	Returns a new set object

setattr()	Sets an attribute (property/method) of an object
slice()	Returns a slice object
sorted()	Returns a sorted list
staticmethod()	Converts a method into a static method
str()	Returns a string object
sum()	Sums the items of an iterator
super()	Returns an object that represents the parent class
tuple()	Returns a tuple
type()	Returns the type of an object
vars()	Returns the __dict__ property of an object
zip()	Returns an iterator, from two or more iterators

---

1. What is the slice operator in Python, and how does it work?

- *Explain the syntax: sequence[start:stop:step] and how it works with lists, tuples, and strings.*

---

2. How can you reverse a string using slicing?

- *Use the slice operator with a negative step to reverse a string:*

```
s = "Python"
print(s[::-1])  # Output: "nohtyP"
```

---

3. What happens if you omit start, stop, or step in slicing?

- *Explain these cases:*
- s = "abcdef"
- print(s[:]) # "abcdef" (Full string)
- print(s[2:]) # "cdef" (From index 2 to end)
- print(s[:4]) # "abcd" (From start to index 4)
- print(s[::2]) # "ace" (Every second character)

---

4. What is the difference between s[:n] and s[-n:]?

- s[:n] → First n characters.
- s[-n:] → Last n characters.
- Example:
- s = "Python"
- print(s[:3]) # "Pyt"
- print(s[-3:]) # "hon"

---

5. How do you extract every alternate character from a string using slicing?

- *Use step size 2:*
- s = "abcdefgh"
- print(s[::2]) # Output: "aceg"

---

6. What happens if start is greater than stop in slicing?

- *Returns an empty string or list, unless using a negative step:*
- s = "Python"
- print(s[4:2]) # "" (empty string)
- print(s[4:2:-1]) # "ot"

---

7. How do you slice a list to get elements from the middle?

- *Extract elements from a list using slicing:*
- nums = [10, 20, 30, 40, 50, 60]

- print(nums[2:5]) # [30, 40, 50]

---

8. How can you remove the first and last characters of a string using slicing?

s = "Python"

print(s[1:-1]) # "ytho" (Removes first and last character)

---

9. What is the behavior of negative step values in slicing?

- *Explain how negative steps work by reversing a string or list:*
- s = "abcdef"
- print(s[::-1])   # "fedcba"
- print(s[4:1:-1]) # "edc"

---

10. How do you create a copy of a list or string using slicing?

- *Use [:] to create a copy:*
- lst = [1, 2, 3, 4]
- new_lst = lst[:] # Creates a new list, not a reference
- print(new_lst)   # [1, 2, 3, 4]

Functions

A function is a **block of code** which **only runs when it is called**.

You can pass data, known as parameters, into a function.

A function can return data as a result.

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

---

1. What is a function in Python, and why is it used?

- *Explain how functions improve code **reusability**, **modularity**, and **readability**.*
- Example:
- def greet(name):
- return f"Hello, {name}!"
- print(greet("Alice"))

---

2. What is the difference between return and print in a function?

- return → Returns a value from a function.
- print → Displays output but does not return a value.
- Example:
- def func():
- return "Hello"
- print(func()) # Output: Hello

---

3. What are *args and **kwargs in Python functions?

- *args → Allows a function to accept any number of positional **arguments**.

- **kwargs → Allows a function to accept any number of **keyword arguments**.
- Example:
- def demo(*args, **kwargs):
- print(args) # Tuple of positional arguments
- print(kwargs) # Dictionary of keyword arguments
-
- demo(1, 2, 3, name="Alice", age=25)

---

4. What is the difference between a normal function and a lambda function?

- *Lambda functions are anonymous, **single-expression functions used for short tasks**.*
- Example:
- **add = lambda x, y: x + y**
- print(add(3, 5)) # Output: 8

---

5. What is recursion? Give an example of a recursive function.

- *A function calling itself to solve smaller subproblems.*
- Example (Factorial):
- def factorial(n):
- if n == 0:
- return 1
- return n * factorial(n - 1)
-
- print(factorial(5)) # Output: 120

---

6. What is the difference between positional arguments and keyword arguments?

- *Positional arguments → Depend on order.*
- *Keyword arguments → Explicitly mention parameter names.*
- Example:
- def greet(name, msg="Hello"):
- return f"{msg}, {name}!"

- print(greet("Alice")) # Hello, Alice!
- print(greet(name="Bob", msg="Hi")) # Hi, Bob!

---

7. What are default arguments in Python functions?

- *A function parameter that has a default value if not provided by the user.*
- Example:
- def power(base, exp=2):
- return base ** exp
-
- print(power(3)) # 9 (default exponent 2)
- print(power(3, 3)) # 27

---

8. What is function scope, and what are local and global variables?

- *Local* → *Defined inside a function and accessible only there.*
- *Global* → *Defined outside a function and accessible everywhere.*
- Example:
- x = 10 # Global variable
-
- def example():
- x = 5 # Local variable
- print(x) # 5 (Local scope)
-
- example()
- print(x) # 10 (Global scope)

---

9. How do you pass a function as an argument to another function?

- Example:
- def square(n):
- return n * n
-
- def apply_func(func, value):
- return func(value)
-
- print(apply_func(square, 4)) # Output: 16

---

10. What is the difference between deepcopy() and copy() in Python functions?

- *copy() → Creates a shallow copy (changes in nested structures affect the copy).*
- *deepcopy() → Creates a deep copy (completely independent copy).*
- Example:
- import copy
-
- lst1 = [[1, 2], [3, 4]]
- lst2 = copy.copy(lst1)
- lst3 = copy.deepcopy(lst1)
-
- lst1[0][0] = 99
- print(lst2) # [[99, 2], [3, 4]] (Shallow copy affected)
- print(lst3) # [[1, 2], [3, 4]] (Deep copy unaffected)

Functions in Python

A **function** in Python is a reusable block of code that performs a specific task. *Functions help in modular programming, making code more readable and maintainable.*

Types of Functions in Python:

1. **Built-in Functions** – Predefined functions like print(), len(), and sum().
2. **User-defined Functions** – Created using the def keyword.

3. **Lambda Functions** – Anonymous functions defined using lambda.

Function Syntax:

```
def function_name(parameters):
    """Docstring (optional)"""
    statement(s)
    return value  # (optional)
```

Example:

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

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

Function Arguments:

- **Positional Arguments** – Based on order.
- **Keyword Arguments** – Explicitly named.
- **Default Arguments** – Have predefined values.
- **Variable-Length Arguments** – *args (tuple) and **kwargs (dictionary).

Scope of Variables:

- **Local** – Defined inside a function.
- **Global** – Accessible throughout the program.

Recursion in Functions:

A function that calls itself, useful for problems like factorial and Fibonacci sequence.

Functions enhance **code reusability**, **efficiency**, and **readability**, making them a crucial part of Python programming.

How to Read a Text File & Writing to a Text File:

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

 # output:

Hello! Welcome to demofile2.txt
This file is for testing purposes.
Good Luck! Now the file has more content!

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the overwriting:
f = open("demofile3.txt", "r")
print(f.read())
```

 # output: Woops! I have deleted the content!

---

1. How do you open a text file in Python?

- *Use the open() function:*
- file = open("example.txt", "r") # Opens file in read mode
- content = file.read()
- file.close()

---

2. What are the different file opening modes in Python?

- "r" – Read (default).
- "w" – Write (creates/truncates).
- "a" – Append.
- "r+" – Read & Write.
- "w+" – Write & Read (truncates).
- "a+" – Append & Read.

---

3. How do you read the entire content of a file?

- *Using read():*
- with open("example.txt", "r") as file:
- content = file.read()
- print(content)

---

4. How do you read a file line by line?

- *Using readline() or readlines():*
- with open("example.txt", "r") as file:
- for line in file:
- print(line.strip()) # Removes extra spaces

---

5. How do you write to a file in Python?

- *Using "w" mode (overwrites existing content):*
- with open("output.txt", "w") as file:
- file.write("Hello, World!\n")

---

6. How do you append data to an existing file?

- *Using "a" mode (adds content without erasing previous data):*
- with open("output.txt", "a") as file:
- file.write("Appending new data.\n")

---

7. What is the difference between w and a modes?

- "w" – ***Overwrites*** existing content or creates a new file.
- "a" – Appends to an existing file or creates a new file.

---

8. How do you read a file as a list of lines?

```
with open("example.txt", "r") as file:
    lines = file.readlines()
    print(lines)  # List of lines
```

---

9. How do you check if a file exists before reading or writing?

- *Using os.path.exists() or Pathlib:*
- import os
-
- if os.path.exists("example.txt"):
- with open("example.txt", "r") as file:
- print(file.read())
- else:
- print("File not found")

---

10. What is the benefit of using with open() over open()?

- *with open() automatically closes the file after execution, avoiding memory leaks.*
- Example:
- with open("example.txt", "r") as file:
- content = file.read() # No need for file.close()

Count & Find

---

1. What is the difference between find() and count() in Python?

- find() → Returns the index of the first occurrence of a substring (or -1 if not found).
- count() → Returns the number of occurrences of a substring.

Example:

```
text = "hello world, hello Python"
print(text.find("hello"))  # Output: 0
print(text.count("hello")) # Output: 2
```

---

2. How does the find() method work in Python?

- *It returns the first occurrence index of a substring in a string.*

```
text = "Python is fun"
print(text.find("is"))  # Output: 7
print(text.find("Java"))  # Output: -1 (not found)
```

---

3. How does the count() method work in Python?

- *It counts occurrences of a substring within a string.*

```
text = "banana banana apple"
print(text.count("banana"))  # Output: 2
```

---

4. What happens if find() doesn’t find the substring?

- It returns -1 instead of raising an error.

```
text = "hello world"
print(text.find("xyz"))  # Output: -1
```

---

5. Can you use count() with a character instead of a substring?

- Yes, count() works with both characters and substrings.

```
text = "hello world"
print(text.count("l"))  # Output: 3
```

---

6. How can you find all occurrences of a substring in a string?

- *Use find() inside a loop to find multiple occurrences.*

```
text = "hello hello world"
start = 0
while (index := text.find("hello", start)) != -1:
    print("Found at:", index)
    start = index + 1
```

---

7. What happens if count() is used with an empty string?

- It returns the number of positions between characters + 1.

```
text = "hello"
print(text.count(""))  # Output: 6 (positions: before h, e, l, l, o, and after o)
```

---

8. How can find() be used to search within a specific range?

- *Use find(substring, start, end) to limit the search range.*

```
text = "hello world, hello Python"
print(text.find("hello", 5))  # Output: 13 (skips first "hello")
```

---

9. Can count() be used with a list?

- *Yes, count() works with lists to count occurrences of elements.*

```
nums = [1, 2, 3, 1, 4, 1]
print(nums.count(1))  # Output: 3
```

---

10. How can you implement find() functionality using index()?

- *index() is similar to find(), but raises an error if the substring is not found.*

```
text = "hello world"
try:
    print(text.index("hello"))  # Output: 0
    print(text.index("xyz"))    # Raises ValueError
except ValueError:
    print("Substring not found")
```

Lambda

A lambda function is a **small anonymous function.**

A lambda function can take any number of arguments, but can only have **one expression.**

Syntax

lambda *arguments* : *expression*

```
x = lambda a, b: a * b
print(x(5, 6))     # 30
```

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))

# output: 22,   33
```

Introduction to Modular Programming

import math, os, more

e.g. print(math.pi)

---

1. What is Modular Programming in Python? Why is it important?

- **Modular programming** is a software design approach where a program is divided into **independent, reusable modules** (files or functions).
- **Advantages:**
    - Increases **code reusability**.
    - Improves **readability & maintainability**.
    - Allows **collaborative development**.
    - Reduces **code duplication**.

Example:

```
# module.py (A separate module)
def greet(name):
    return f"Hello, {name}!"
# main.py (Importing the module)
import module
print(module.greet("Alice"))  # Output: Hello, Alice!
```

---

2. How do you create and import a module in Python?

- A **module** is a Python file (.py) that contains **functions, classes, or variables**.
- You can **import** it using import or from module import function.

Example:

```
# math_utils.py
def square(num):
    return num * num
# main.py
```

```
import math_utils
print(math_utils.square(5))  # Output: 25
```

---

3. What is the difference between import module and from module import function?

- import module → Imports the **whole module** and requires **module name** to access functions.
- import math_utils
- print(math_utils.square(4)) # Output: 16
- from module import function → Imports **only a specific function** (no need to use module name).
- from math_utils import square
- print(square(4)) # Output: 16
- from module import * → Imports **all functions**, but it's **not recommended** (can cause name conflicts).

---

4. What are Python packages? How are they different from modules?

- A **module** is a **single** Python file (.py).
- A **package** is a **collection of modules** inside a directory containing an __init__.py file.

Example:

```
mypackage/          ← Package (folder)
├── __init__.py     ← Required for a package
├── math_utils.py   ← Module 1
├── string_utils.py← Module 2
```

Importing from a package:

```
from mypackage.math_utils import square
print(square(3))  # Output: 9
```

---

5. What is the purpose of __name__ == "__main__" in modular programming?

- Used to check **if a script is running directly or being imported as a module**.
- If __name__ == "__main__", the script runs; otherwise, it behaves as a module.

Example:

```
# mymodule.py
def greet():
    print("Hello!")


if __name__ == "__main__":
    greet()  # Runs only if executed directly
```

Running as a script:

```
$ python mymodule.py
```

Output: Hello!

Importing in another script:

```
import mymodule  # Doesn't print "Hello!"
```

*** we create our own model very easily….

Optional Parameters / Adv function

---

1. What are optional parameters in Python?

- **Optional parameters** are **function arguments** that have a **default value**, making them **optional** to pass when calling the function.
- **Example:**
- def greet(name="Guest"):
- return f"Hello, {name}!"
-
- print(greet()) # Output: Hello, Guest!
- print(greet("Alice")) # Output: Hello, Alice!

---

2. How do you define a function with optional parameters?

- **Use default values in function parameters.**
- def power(base, exponent=2):
- return base ** exponent
-
- print(power(3)) # Output: 9 (default exponent=2)
- print(power(3, 3)) # Output: 27 (overrides default)

---

3. Can optional parameters be placed before required parameters?

- **No**, required parameters must come **before** optional ones.
- Incorrect:
- def greet(name="Guest", age): # SyntaxError
- return f"{name} is {age} years old"
- Correct:
- def greet(age, name="Guest"):
- return f"{name} is {age} years old"

---

4. How can optional parameters be used with None as a default?

- **Use None when a parameter's default needs to be dynamically assigned.**
- def greet(name=None):
- if name is None:
- name = "Guest"
- return f"Hello, {name}!"
-
- print(greet()) # Output: Hello, Guest!
- print(greet("Alice")) # Output: Hello, Alice!

---

5. What happens if a function has multiple optional parameters?

- **Python assigns values based on position, unless keyword arguments are used.**
- def describe_pet(animal="dog", name="Buddy"):
- return f"{name} is a {animal}."
-
- print(describe_pet()) # Output: Buddy is a dog.
- print(describe_pet("cat")) # Output: Buddy is a cat.
- print(describe_pet(name="Charlie")) # Output: Charlie is a dog.

---

6. How can optional parameters be used with *args and **kwargs?

- *args → Accepts multiple **positional arguments**.
- **kwargs → Accepts multiple **keyword arguments**.
- def greet(*names, message="Hello"):
- for name in names:
- print(f"{message}, {name}!")
-
- greet("Alice", "Bob", message="Hi")

- # Output:
- # Hi, Alice!
- # Hi, Bob!

---

7. What is the difference between None as a default value and an empty list ([])?

- Using **a mutable object** like [] as a default can lead to **unexpected behavior**.
- Example:
- def add_item(item, lst=[]): # Avoid mutable defaults!
-     lst.append(item)
-     return lst
-
- print(add_item("Apple")) # Output: ['Apple']
- print(add_item("Banana")) # Output: ['Apple', 'Banana'] (not a fresh list)
- Correct approach:
- def add_item(item, lst=None):
-     if lst is None:
-         lst = []
-     lst.append(item)
-     return lst

---

8. How can you call a function with both optional and required parameters using keyword arguments?

- **Mix positional and keyword arguments for clarity.**
- def greet(name, greeting="Hello"):
-     return f"{greeting}, {name}!"
-
- print(greet("Alice", greeting="Hi")) # Output: Hi, Alice!

---

9. Can None be an explicit argument even when an optional parameter has a default value?

- **Yes, passing None overrides the default.**
- def welcome(user="Guest"):
- return f"Welcome, {user}!"
-
- print(welcome(None)) # Output: Welcome, None!

---

10. What is the best practice for handling optional parameters in Python functions?

- **Best Practices:**
  - **Use None instead of mutable objects ([], {}, etc.).**
  - **Use keyword arguments (**kwargs) for flexibility.**
  - **Document default values in function docstrings.**

Error Handling

The ***try*** block lets you test a block of code for errors.

The ***except*** block lets you handle the error.

The ***finally*** block lets you execute code, regardless of the result of the try- and except blocks.

---

1. What is Exception Handling in Python? Why is it important?

- **Exception Handling** is a mechanism to handle runtime errors and prevent program crashes.
- It ensures **graceful error recovery** and **better debugging**.
- **Example:**
- try:
- x = 10 / 0 # Division by zero error
- except ZeroDivisionError:
- print("Cannot divide by zero!")

---

2. What is the difference between Syntax Errors and Exceptions?

- **Syntax Errors:**
  - Occur when Python **fails to parse** the code.
  - Example:
  - if True # Missing colon (SyntaxError)
  - print("Hello")
- **Exceptions:**
  - Occur during runtime (e.g., division by zero, accessing an invalid index).
  - Example:
  - print(10 / 0) # ZeroDivisionError

---

3. What are try, except, finally, and else in Python?

- try: Code that **may raise an error**.

- except: Block to **handle exceptions**.
- else: Runs **if no exception** occurs. {Optional}
- finally: Always runs, whether an error occurs or not. [Default it's print].

Example:

```
try:
    x = int("abc")  # ValueError
except ValueError:
    print("Invalid input!")
else:
    print("Success!")  # Runs if no error.
finally:
    print("Execution complete.")  # Always runs
```

---

4. How can you handle multiple exceptions in Python?

- Use **multiple except blocks** or handle multiple errors in a **single except block**.

```
try:
    x = int("abc")
except (ValueError, TypeError) as e:
    print(f"Error: {e}")
```

---

5. What is the difference between Exception and specific exception classes?

- **Exception** is the **base class** for all exceptions.
- **Specific exceptions** (e.g., ValueError, ZeroDivisionError) provide better error handling.
- **Example:**
- try:
- print(1 / 0)
- except Exception as e: # Catches all exceptions
- print(f"Error: {e}")

---

6. How do you create a custom exception in Python?

- Inherit from the Exception class and define your own exception.

```
class CustomError(Exception):
    pass


try:
    raise CustomError("Something went wrong!")
except CustomError as e:
    print(f"Custom Exception: {e}")
```

---

7. What is the purpose of the raise keyword in Python?

- The raise statement is used to **manually trigger an exception**.
- **Example:**
- def check_age(age):
- if age < 18:
- raise ValueError("Age must be 18 or above!")
- return "Access granted"
-
- print(check_age(16)) # Raises ValueError

---

8. What is assert in Python? How is it different from raise?

- **assert** is used for debugging; it checks conditions and raises an AssertionError if false.
- **Example:**
- x = 5
- assert x > 10, "Value must be greater than 10" # Raises AssertionError
- assert is mainly used for **debugging**, whereas raise is for **manual exception handling**.

---

9. What is exception chaining in Python? (raise from)

- Exception chaining helps trace errors **when one exception is raised inside another**.

```
try:
    raise ValueError("Original error")
except ValueError as e:
    raise RuntimeError("New error") from e
```

- This **preserves the original exception** for debugging.

---

10. What are common built-in exceptions in Python?

- **ZeroDivisionError** → Division by zero
- **ValueError** → Invalid value type
- **IndexError** → Accessing an invalid index
- **KeyError** → Accessing a non-existent dictionary key
- **TypeError** → Invalid operation on data types

Example:

```
try:
    my_list = [1, 2, 3]
    print(my_list[5])  # IndexError
except IndexError as e:
    print(f"Error: {e}")
```

```
Name = input("Enter the User Name: ")
try:
    num = int(Name)
    print(num)
except:
    print("Enter Valid name")
```

Global vs Local Variables

---

1. What is the difference between global and local variables in Python?

- **Local Variable:** Defined inside a function and accessible only within that function.
- **Global Variable:** Defined outside any function and accessible throughout the program.

Example:

```
x = 10  # Global variable

def my_function():
    y = 5  # Local variable
    print(y)

my_function()
print(x)  # Works fine
print(y)  # Error: y is not defined (local scope)
```

---

2. Can a local variable have the same name as a global variable?

- **Yes**, but the local variable **overrides** the global one within the function.

```
x = 10  # Global variable

def my_function():
    x = 5  # Local variable (shadows global x)
    print(x)  # Output: 5

my_function()
print(x)  # Output: 10 (Global x remains unchanged)
```

---

3. How do you modify a global variable inside a function?

- Use the global keyword to modify a global variable inside a function.

```
x = 10

def modify_global():
    global x
    x = 20  # Modifies the global x

modify_global()
print(x)  # Output: 20
```

---

4. What happens if you try to modify a global variable inside a function without the global keyword?

- Python **creates a new local variable** instead of modifying the global one.

```
x = 10

def modify():
    x = 20  # Creates a new local variable
    print(x)  # Output: 20

modify()
print(x)  # Output: 10 (global x remains unchanged)
```

---

5. What is the nonlocal keyword, and how is it different from global?

- nonlocal is used inside a **nested function** to modify a **variable from the enclosing function's scope** (not global scope).

```
def outer():
    x = 10  # Enclosing variable
```

```
    def inner():
        nonlocal x
        x = 20  # Modifies outer function's x

    inner()
    print(x)  # Output: 20

outer()
```

---

6. What is the scope resolution order in Python?

- Python follows the **LEGB rule** (Local → Enclosing → Global → Built-in).
- **Example:**
- x = "global"
-
- def outer():
- x = "enclosing"
- def inner():
- x = "local"
- print(x) # Output: local
- inner()
- outer()
- print(x) # Output: global

---

7. Can global variables be accessed inside a function without using global?

- **Yes,** but you cannot **modify** them without the global keyword.

```
x = 10

def access_global():
    print(x)  # Works fine
```

access_global()

- **But modifying without global gives an error:**
- x = 10
-
- def modify_global():
- x += 5 # UnboundLocalError (Python treats x as local)

---

8. Are global variables bad practice? When should you use them?

- Global variables can cause **unexpected side effects** and make debugging difficult.
- Best practices:
  - Use global variables **only for constants** or configuration settings.
  - Use **function parameters** and **return values** instead of modifying global variables.

---

9. How can you avoid global variables while keeping shared state?

- Use **function parameters** instead of modifying global variables.
- Use **class attributes** if data needs to persist.

```
class Counter:
    def __init__(self):
        self.count = 0  # Instance variable

    def increment(self):
        self.count += 1
        return self.count

c = Counter()
print(c.increment())  # Output: 1
print(c.increment())  # Output: 2
```

10. How do built-in global functions behave with local variables?

- Python's **built-in functions** are in the global scope and can be overridden by local variables.
- Example:
- def print():
- return "Overridden!"
-
- print(print()) # TypeError: 'str' object is not callable

Object Oriented Programming

Class and Object

A class is a blueprint for creating objects. It defines properties (attributes) and behaviors (methods) that its objects will have.

Object:

An object is an instance of a class. Each object has its own unique data, but they all share the class structure.

Example:

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

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

# Creating objects
car1 = Car("Toyota", "Camry")
car2 = Car("Honda", "Civic")

car1.display()  # Output: Car: Toyota Camry
car2.display()  # Output: Car: Honda Civic
```

---

1. What is the difference between a class and an object?

Answer:

- A class is a blueprint for creating objects, whereas an object is an instance of a class.
- The class defines attributes and methods, while the object stores actual data.

Example:

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

```
dog1 = Dog("Buddy")  # Object created from Dog class
```

---

2. What is the purpose of the __init__() method?

Answer:

- The __init__() method is a constructor that initializes the object's attributes when an object is created.
- It is called automatically when an object is instantiated.

Example:

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

p1 = Person("Alice", 25)  # Calls __init__() method
```

---

3. What is self in Python classes?

Answer:

- self refers to the current instance of the class and allows access to attributes and methods.
- It must be the first parameter of instance methods.

Example:

```
class Student:
    def __init__(self, name):
        self.name = name  # `self.name` is the instance variable

    def greet(self):
        print(f"Hello, my name is {self.name}")

s = Student("John")
s.greet()  # Output: Hello, my name is John
```

---

4. What are instance variables and class variables?

Answer:

- Instance variables are specific to each object (defined inside __init__()).
- Class variables are shared among all instances of a class.

Example:

```
class Employee:
    company = "Google"  # Class variable

    def __init__(self, name):
        self.name = name  # Instance variable

e1 = Employee("Alice")
e2 = Employee("Bob")

print(e1.company)  # Output: Google
print(e2.company)  # Output: Google
```

---

5. What is method overloading in Python?

Answer:

- Python does not support traditional method overloading (same method name, different parameters).
- However, it can be simulated using default arguments or *args.

Example:

```
class Math:
    def add(self, a, b, c=0):
        return a + b + c

m = Math()
print(m.add(2, 3))       # Output: 5
print(m.add(2, 3, 4))    # Output: 9
```

---

6. What is method overriding?

Answer:

- Method overriding allows a subclass to redefine a method of its parent class.

Example:

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

d = Dog()
d.speak()  # Output: Dog barks
```

---

7. What is a static method in Python?

Answer:

- A static method does not use self and does not access instance attributes.
- It is defined using @staticmethod.

Example:

```
class Utility:
    @staticmethod
    def add(a, b):
        return a + b

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

---

8. What is the difference between class methods and instance methods?

Answer:

Feature	Instance Method (self)	Class Method (cls)
Uses self?	Yes	No
Uses cls?	No	Yes
Can modify instance variables?	Yes	No
Can modify class variables?	No	Yes
Decorator	None	@classmethod

Example:

```
class Sample:
    class_var = 10

    def instance_method(self):
        return "Instance Method"

    @classmethod
    def class_method(cls):
        return f"Class Method: {cls.class_var}"

obj = Sample()
print(obj.instance_method())  # Output: Instance Method
print(Sample.class_method())  # Output: Class Method: 10
```

---

9. What is the difference between deep copy and shallow copy?

Answer:

- Shallow Copy: Copies only references to objects, not the objects themselves.
- Deep Copy: Creates a new independent copy of objects.

Example:

```
import copy

list1 = [[1, 2], [3, 4]]
shallow = copy.copy(list1)
```

```
deep = copy.deepcopy(list1)


list1[0][0] = 99
print(shallow)  # [[99, 2], [3, 4]] (Changed)
print(deep)     # [[1, 2], [3, 4]] (Unchanged)
```

---

10. What is multiple inheritance in Python?

Answer:

- Multiple inheritance allows a class to inherit from multiple parent classes.
- Python follows the Method Resolution Order (MRO) to handle conflicts.

Example:

```
class A:
    def show(self):
        print("Class A")


class B:
    def show(self):
        print("Class B")


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


obj = C()
obj.show()  # Output: Class A (follows MRO)
```

Final Thoughts

These are some of the most commonly asked interview questions on classes and objects in Python. Understanding these concepts with examples will help you confidently tackle OOP-based questions in coding interviews. 🚀

What is Inheritance

Inheritance is an object-oriented programming (OOP) concept where one class (child/derived class) can inherit properties and methods from another class (parent/base class). It promotes code reuse and hierarchy management.

Types of Inheritance in Python

1. Single Inheritance → One class inherits from another.
2. Multiple Inheritance → A class inherits from more than one class.
3. Multilevel Inheritance → A derived class inherits from another derived class.
4. Hierarchical Inheritance → Multiple derived classes inherit from a single base class.
5. Hybrid Inheritance → A combination of multiple inheritance types.

---

1. What is inheritance in Python? Why is it used?

Answer:
Inheritance allows a child class to reuse the properties and methods of a parent class, reducing code duplication. It also supports polymorphism and code organization.

Example:

```
class Animal:
    def speak(self):
        print("Animal makes a sound")


class Dog(Animal):  # Inheriting from Animal
    def bark(self):
        print("Dog barks")


d = Dog()
d.speak()  # Output: Animal makes a sound
d.bark()   # Output: Dog barks
```

---

2. What are the types of inheritance in Python?

Answer:

- Single Inheritance → One child class inherits from one parent class.

- Multiple Inheritance → One child class inherits from multiple parent classes.
- Multilevel Inheritance → Child class inherits from a derived class.
- Hierarchical Inheritance → Multiple child classes inherit from a single parent class.
- Hybrid Inheritance → A mix of multiple inheritance types.

---

3. What is single inheritance? Provide an example.

Answer:
Single inheritance allows a derived class to inherit from a single base class.

Example:

```
class Parent:
    def show(self):
        print("Parent class")


class Child(Parent):
    def display(self):
        print("Child class")


c = Child()
c.show()    # Output: Parent class
c.display() # Output: Child class
```

---

4. What is multiple inheritance? Give an example.

Answer:
Multiple inheritance allows a class to inherit from more than one parent class.

Example:

```
class A:
    def methodA(self):
        print("Method from Class A")


class B:
    def methodB(self):
```

```
        print("Method from Class B")

class C(A, B):  # Multiple Inheritance
    def methodC(self):
        print("Method from Class C")

obj = C()
obj.methodA()  # Output: Method from Class A
obj.methodB()  # Output: Method from Class B
obj.methodC()  # Output: Method from Class C
```

---

5. What is multilevel inheritance? Provide an example.

Answer:
Multilevel inheritance occurs when a derived class inherits from another derived class.

Example:

```
class Grandparent:
    def grandparent_method(self):
        print("Grandparent method")

class Parent(Grandparent):
    def parent_method(self):
        print("Parent method")

class Child(Parent):
    def child_method(self):
        print("Child method")

c = Child()
c.grandparent_method()  # Output: Grandparent method
c.parent_method()       # Output: Parent method
c.child_method()        # Output: Child method
```

---

6. What is hierarchical inheritance? Provide an example.

Answer:
Hierarchical inheritance occurs when multiple child classes inherit from a single parent class.

Example:

```
class Parent:
    def common_method(self):
        print("Common method from Parent")


class Child1(Parent):
    def child1_method(self):
        print("Child1 method")


class Child2(Parent):
    def child2_method(self):
        print("Child2 method")


c1 = Child1()
c2 = Child2()


c1.common_method()  # Output: Common method from Parent
c2.common_method()  # Output: Common method from Parent
```

---

7. What is the Method Resolution Order (MRO) in Python?

Answer:
MRO defines the order in which methods are inherited when multiple inheritance is used. It follows the C3 Linearization Algorithm.

Example:

```
class A:
    def show(self):
        print("Class A")
```

```
class B(A):
    def show(self):
        print("Class B")

class C(A):
    def show(self):
        print("Class C")

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

d = D()
d.show()  # Output: Class B (Follows MRO: D → B → C → A)
```

◇ Use D.mro() to check MRO:

```
print(D.mro())
# Output: [<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

---

8. What is the super() function in Python?

Answer:

- super() allows calling a method from the parent class inside the child class.
- It is useful for method overriding.

Example:

```
class Parent:
    def show(self):
        print("Parent class")

class Child(Parent):
    def show(self):
        super().show()  # Calls parent class method
```

```
    print("Child class")

c = Child()
c.show()
```

Output:

Parent class

Child class

---

9. What is method overriding? How does it work?

Answer:

- Method overriding allows a child class to provide a different implementation of a method from the parent class.
- It is useful for polymorphism.

Example:

```
class Parent:
    def show(self):
        print("This is Parent class")

class Child(Parent):
    def show(self):  # Overriding the method
        print("This is Child class")

c = Child()
c.show()  # Output: This is Child class
```

---

10. What is hybrid inheritance? Provide an example.

Answer:
Hybrid inheritance is a combination of multiple types of inheritance.

Example:

```
class A:
    def methodA(self):
```

```
        print("Class A")

class B(A):
    def methodB(self):
        print("Class B")

class C(A):
    def methodC(self):
        print("Class C")

class D(B, C):  # Hybrid Inheritance
    def methodD(self):
        print("Class D")

d = D()
d.methodA()  # Output: Class A
d.methodB()  # Output: Class B
d.methodC()  # Output: Class C
d.methodD()  # Output: Class D
```

Final Thoughts

- Inheritance is a core concept in OOP that improves code reusability and polymorphism.
- Understanding method overriding, MRO, super(), and different inheritance types is essential for interviews.
- Practice coding questions on OOP and design patterns to strengthen your understanding.

Method overloading means defining multiple methods in a class with the same name but different parameters.

Both (Method Overloading & Method Overriding) have the same method name in Java.

- Method Overloading: Different parameters (different number or type).
- Method Overriding: Same parameters (must match exactly).

Polymorphism:

Polymorphism means "many forms". It allows objects of different classes to be treated as objects of a common base class. The main goal of polymorphism is to allow one interface to be used for different types.

Types of Polymorphism in Python

1. Method Overloading (Not directly supported in Python but can be implemented using default arguments or *args).
2. Method Overriding (Same method name in parent and child class, but different behavior).
3. Operator Overloading (Using +, -, *, etc., with custom objects).
4. Duck Typing (If an object behaves like a specific type, it is treated as that type).

---

1. What is polymorphism in Python? Why is it used?

Answer:
Polymorphism allows different classes to be treated uniformly through a common interface. It promotes flexibility, reusability, and scalability in OOP.

Example:

```
class Cat:
    def speak(self):
        return "Meow"

class Dog:
    def speak(self):
        return "Bark"

def animal_sound(animal):
    print(animal.speak())

cat = Cat()
dog = Dog()

animal_sound(cat)  # Output: Meow
```

animal_sound(dog) # Output: Bark

🔹 The function animal_sound() works with both Cat and Dog objects, demonstrating polymorphism.

---

2. What is method overloading? Is it supported in Python?

Answer:

- Method Overloading means *defining multiple methods with the same name but different parameters.*
- *Python does not support method overloading* like **Java/C++.** Instead, it can be simulated using *default arguments or *args.*

Example:

```
class Math:
    def add(self, a, b, c=0):
        return a + b + c

m = Math()
print(m.add(2, 3))       # Output: 5
print(m.add(2, 3, 4))    # Output: 9
```

🔹 Here, c has a default value, making method overloading possible.

---

3. What is method overriding? How is it different from method overloading?

Answer:

- Method Overriding occurs when a child class provides a new implementation for a method already defined in the parent class.
- Method **Overloading** is defining *multiple methods with the same name but different parameters* (not directly supported in Python).

Example of Method Overriding:

```
class Parent:
    def show(self):
        print("Parent class")

class Child(Parent):
```

```
    def show(self):  # Overriding the method
        print("Child class")


c = Child()
c.show()  # Output: Child class
```

- The show() method in the Child class overrides the show() method of the Parent class.

---

4. What is operator overloading? Give an example.

Answer:

- Operator Overloading allows built-in operators (+, -, *, /, etc.) to be redefined for user-defined objects.
- It is done using magic methods (e.g., __add__, __sub__, __mul__).

Example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

p1 = Point(2, 3)
p2 = Point(4, 5)
p3 = p1 + p2  # Calls __add__ method
print(p3.x, p3.y)  # Output: 6 8
```

- The + operator now works with Point objects, adding their x and y values.

---

5. What is duck typing in Python? Give an example.

Answer:

- Duck Typing means that an ***object's behavior determines its type***, rather than inheritance.

- If an object has the required method, it can be used, regardless of its actual class.

Example:

```
class Bird:
    def fly(self):
        print("Bird is flying")


class Airplane:
    def fly(self):
        print("Airplane is flying")


def perform_flight(obj):
    obj.fly()


bird = Bird()
airplane = Airplane()


perform_flight(bird)     # Output: Bird is flying
perform_flight(airplane) # Output: Airplane is flying
```

- The perform_flight() function works with any object that has a fly() method, demonstrating duck typing.

---

6. What is function polymorphism? Provide an example.

Answer:
Function polymorphism means that the same function name can be used for different types.

Example:

```
print(len("Hello"))      # Output: 5 (String)
print(len([1, 2, 3, 4])) # Output: 4 (List)
print(len({1: "A", 2: "B"})) # Output: 2 (Dictionary)
```

- The len() function works on different data types, showing polymorphism.

7. How does polymorphism work with inheritance?

Answer:

- Polymorphism allows child classes to override methods while still being treated as instances of the parent class.

Example:

```
class Vehicle:
    def fuel_type(self):
        return "Petrol or Diesel"

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

v = Vehicle()
e = ElectricCar()

print(v.fuel_type())  # Output: Petrol or Diesel
print(e.fuel_type())  # Output: Electric
```

- The fuel_type() method is overridden in ElectricCar, providing a different output.

---

8. How do abstract classes support polymorphism?

Answer:

- Abstract classes define common methods but leave implementation to derived classes.
- This enforces polymorphism as all child classes must implement the abstract method.

Example:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass
```

```
class Dog(Animal):
    def speak(self):
        return "Bark"

class Cat(Animal):
    def speak(self):
        return "Meow"

d = Dog()
c = Cat()
print(d.speak())  # Output: Bark
print(c.speak())  # Output: Meow
```

- The speak() method is abstract in Animal, ensuring all child classes define it.

---

9. Can a class be polymorphic without inheritance?

Answer:
Yes! Polymorphism can be achieved without inheritance through duck typing and function polymorphism.

Example:

```
class Car:
    def start(self):
        return "Car is starting"

class Computer:
    def start(self):
        return "Computer is booting"

def start_device(device):
    print(device.start())
```

```
car = Car()
computer = Computer()


start_device(car)       # Output: Car is starting
start_device(computer)  # Output: Computer is booting
```

🔹 start_device() works with both Car and Computer, demonstrating polymorphism without inheritance.

---

10. What are the benefits of polymorphism in OOP?

Answer:

1. *Code Reusability* – The same function works with different types.
2. *Flexibility* – Objects can be replaced without modifying code.
3. *Scalability* – Easily extendable to new classes.
4. Encapsulation & Abstraction – Enhances maintainability.

Final Thoughts

- Polymorphism makes OOP more flexible, reusable, and scalable.
- Understanding overloading, overriding, duck typing, and operator overloading is key for interviews.

Encapsulation:

Encapsulation is one of the fundamental concepts of Object-Oriented Programming (OOP). It refers to binding data (variables) and methods (functions) into a single unit (class) and restricting direct access to some of the object's details.

Key Features of Encapsulation:

1. Data Hiding: Prevents direct access to object data, ensuring better security.
2. Access Control: Uses public, protected, and private members to control visibility.
3. Getter & Setter Methods: Provides controlled access to private attributes.
4. Better Maintainability & Reusability: Helps manage and organize complex code.

Access Modifiers in Python

Python does not have strict access modifiers like public, private, and protected in Java or C++. Instead, it uses naming conventions:

Access Modifier	Syntax	Description
Public	var	Accessible anywhere
Protected	**_var**	Intended for subclass use
Private	***__var***	Name-mangled (cannot be accessed directly)

---

1. What is encapsulation in Python? Why is it important?

Answer:
Encapsulation is the process of restricting direct access to some components of an object and only exposing necessary functionalities. It **improves data security, modularity, and maintainability.**

Example:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private variable

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
```

```
        return self.__balance


account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
# print(account.__balance)  # Error: AttributeError
```

- Here, __balance is private, so it cannot be accessed directly outside the class.

---

2. How do you implement private variables in Python?

Answer:
In Python, private variables are declared by prefixing two underscores (__) before the variable name.

Example:

```
class Car:
    def __init__(self, brand):
        self.__brand = brand  # Private variable

    def get_brand(self):  # Getter method
        return self.__brand


car = Car("Tesla")
print(car.get_brand())  # Output: Tesla
# print(car.__brand)  # Error: AttributeError
```

- The **variable __brand is name-mangled, meaning it cannot be accessed directly**.

---

3. What is the difference between public, protected, and private members?

Answer:

Modifier	Syntax	Accessibility
Public	var	Accessible anywhere
Protected	_var	Accessible within the class & subclasses

Modifier	Syntax	Accessibility
Private	__var	Not accessible outside the class (name-mangled)

Example:

```
class Example:
    def __init__(self):
        self.public = "Public"
        self._protected = "Protected"
        self.__private = "Private"

obj = Example()
print(obj.public)      # ✅ Accessible
print(obj._protected)  # ✅ Accessible (but should be used within subclasses)
# print(obj.__private)  # ❌ Error: AttributeError
```

- Only the private variable is truly hidden from outside access.

---

4. How do you access private variables in Python?

Answer:
Private variables cannot be accessed directly, but they can be accessed using **getter methods** or name mangling.

Example (Using Getter Method):

```
class Employee:
    def __init__(self, salary):
        self.__salary = salary  # Private variable

    def get_salary(self):  # Getter method
        return self.__salary

emp = Employee(50000)
print(emp.get_salary())  # Output: 50000
```

- The get_salary() method allows controlled access to __salary.

Example (Using Name Mangling – Not Recommended):

```
print(emp._Employee__salary)  # Output: 50000 (Not recommended)
```

- Name mangling (_ClassName__variable) bypasses encapsulation, which defeats its purpose.

---

5. What is the use of getter and setter methods in encapsulation?

Answer:

- **Getters allow reading** private variables.
- Setters allow modifying private variables with validation.

Example:

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

    def get_name(self):  # Getter
        return self.__name

    def set_name(self, new_name):  # Setter
        if len(new_name) > 2:
            self.__name = new_name
        else:
            print("Invalid name!")

student = Student("John")
print(student.get_name())  # Output: John

student.set_name("Jo")  # Output: Invalid name!
student.set_name("Mike")
print(student.get_name())  # Output: Mike
```

- The setter prevents invalid input, improving data integrity.

---

6. Can we access private variables outside the class?

Answer:
No, private variables cannot be accessed directly. However, they can be accessed using getter methods or name mangling.

---

7. What happens if we try to modify a private variable outside the class?

Answer:
Python will not modify the actual private variable; instead, it will create a new public variable.

Example:

```
class Demo:
    def __init__(self):
        self.__hidden = "Secret"


obj = Demo()
obj.__hidden = "New Value"  # This creates a new variable, does not change the original
print(obj.__hidden)  # Output: New Value
print(obj._Demo__hidden)  # Output: Secret (Original value)
```

- The actual private variable remains unchanged.

---

8. What is the difference between encapsulation and abstraction?

Answer:

Feature	Encapsulation	Abstraction
Purpose	Hides data	Hides implementation details
Focus	Data security	Reducing complexity
Example	Private variables	Abstract classes/methods

---

9. How does encapsulation improve code security?

Answer:

- Prevents accidental modification of critical data.
- Restricts unauthorized access using private variables.

- Ensures controlled access via getter and setter methods.

---

10. How does encapsulation improve modularity and reusability?

Answer:
Encapsulation allows a class to be used without exposing its internal details.

Example:

```
class Database:
    def __init__(self):
        self.__password = "secure_password"  # Private variable

    def connect(self):
        print("Connecting to database...")

db = Database()
db.connect()  # Output: Connecting to database
# print(db.__password)  # Error: AttributeError
```

- The password is hidden, ensuring security.

Final Thoughts

- Encapsulation improves security, modularity, and reusability.
- Always use getter and setter methods to access private data.
- Avoid name mangling unless absolutely necessary.
- Encapsulation is a best practice in OOP to prevent unintended data modification.

Abstraction:

Abstraction is one of the fundamental concepts of Object-Oriented Programming (OOP). It ***hides the implementation details and only exposes essential functionalities to the user***. It allows developers to focus on what an object does rather than how it does it.

Key Features of Abstraction:

1. *Hides Complex Details:* Users interact with an interface rather than the underlying code.
2. Achieved Using Abstract Classes and Methods: In Python, abstraction is implemented using the ABC (Abstract Base Class) module.
3. *Improves Code Maintainability:* Reduces complexity and makes the code more modular.
4. Encourages Loose Coupling: The implementation can be changed without affecting other parts of the code.

How to Achieve Abstraction in Python?

Abstraction in Python is achieved using abstract classes and abstract methods.

- **Abstract Class**: A class that cannot be instantiated and contains at least one abstract method.
- **Abstract Method:** A method that has **no implementation in the base class but must be implemented in derived classes.**

Python provides the ABC (Abstract Base Class) module to create abstract classes.

Example of Abstraction in Python

```
from abc import ABC, abstractmethod


class Vehicle(ABC):  # Abstract class
    @abstractmethod
    def start_engine(self):  # Abstract method
        pass


    @abstractmethod
    def stop_engine(self):
        pass
```

```
class Car(Vehicle):  # Concrete class
    def start_engine(self):
        print("Car engine started")

    def stop_engine(self):
        print("Car engine stopped")

# obj = Vehicle()  # ❌ Error: Cannot instantiate abstract class
car = Car()
car.start_engine()  # ☑ Output: Car engine started
car.stop_engine()   # ☑ Output: Car engine stopped
```

- Vehicle is an abstract class that defines a blueprint for its subclasses.
- The Car class implements the abstract methods, making it a concrete class.

---

1. What is abstraction in Python? Why is it important?

Answer:
Abstraction is an OOP concept that *hides implementation details and only exposes the necessary functionalities to the user.*

Importance:

- *Reduces complexity.*
- *Improves security by hiding implementation details.*
- *Increases code reusability and scalability.*

---

2. How do you achieve abstraction in Python?

Answer:
In Python, abstraction is achieved using abstract classes and abstract methods from the ABC module.

Example:

```
from abc import ABC, abstractmethod

class Animal(ABC):  # Abstract class
    @abstractmethod
```

```
    def make_sound(self):
        pass


class Dog(Animal):  # Concrete class
    def make_sound(self):
        return "Woof!"


dog = Dog()
print(dog.make_sound())  # Output: Woof!
```

- The Animal class is abstract, and Dog provides an implementation for make_sound().

---

3. Can we create an instance of an abstract class?

Answer:
No, we cannot instantiate an abstract class. If we try, Python will raise a TypeError.

```
animal = Animal()  # ❌ TypeError: Can't instantiate abstract class Animal
```

- Abstract classes only provide a blueprint for subclasses.

---

4. What is the difference between abstraction and encapsulation?

Answer:

Feature	Abstraction	Encapsulation
Purpose	Hides implementation details	Hides data from unauthorized access
Focus	Exposing only relevant details	Protecting data from modification
How?	Achieved via abstract classes	**Achieved using access modifiers** (private, protected)
Example	Using abstract classes & methods	Using private variables and getter/setter methods

---

5. Why do we use the ABC module in Python?

Answer:
The ABC module (Abstract Base Classes) provides a way to define abstract classes and methods in Python.

Example:

```
from abc import ABC, abstractmethod


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


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


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


circle = Circle(5)
print(circle.area())  # Output: 78.5
```

- The Shape class defines an abstract method area(), which must be implemented by Circle.

---

6. What happens if a subclass does not implement all abstract methods?

Answer:
If a subclass does not implement all abstract methods, it remains an abstract class and cannot be instantiated.

```
class Square(Shape):  # ❌ Error: Doesn't implement area()
    pass


sq = Square()  # ❌ TypeError
```

- Fix: Implement the missing method.

---

7. Can an abstract class have a constructor in Python?

Answer:
Yes, an abstract class can have a constructor and regular methods.

```
class Employee(ABC):
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    @abstractmethod
    def work(self):
        pass

class Manager(Employee):
    def work(self):
        return f"{self.name} manages the team."

manager = Manager("Alice", 70000)
print(manager.work())  # Output: Alice manages the team.
```

◇ Abstract classes can have constructors, but the abstract methods must be implemented in the subclass.

---

8. Can an abstract class have implemented methods?

Answer:
Yes, an abstract class can have both abstract and concrete (implemented) methods.

```
class Animal(ABC):
    def sound(self):  # Concrete method
        return "Some sound"

    @abstractmethod
    def make_sound(self):
        pass
```

◇ Subclasses must override the abstract method but can use the concrete method.

---

9. How does abstraction improve code maintainability?

Answer:

- *Separates logic from implementation.*
- *Reduces code duplication by defining common behavior in abstract classes.*
- *Encourages modularity, making it easier to update and manage code.*

---

10. Give a real-world example of abstraction in Python.

Answer:

Example: ATM Machine

An ATM hides internal processing (such as authentication**,** balance checking) and only provides a user interface.

```
from abc import ABC, abstractmethod


class ATM(ABC):
    @abstractmethod
    def withdraw(self, amount):
        pass


class MyBankATM(ATM):
    def withdraw(self, amount):
        print(f"Withdrawing {amount} from MyBankATM")


atm = MyBankATM()
atm.withdraw(1000)  # Output: Withdrawing 1000 from MyBankATM
```

◇ The ATM class defines an abstract method withdraw(), which must be implemented by the MyBankATM class.

Final Thoughts

- Abstraction hides implementation details and only provides essential functionalities.

- It is achieved using abstract classes and methods (ABC module).
- Abstract classes cannot be instantiated, and subclasses must implement all abstract methods.

Introduction to Objects

---

1. What is an object in Python?

- An **object** is an **instance of a class** that contains both **data (attributes)** and **behavior (methods)**.
- Everything in Python is an object, including **integers, strings, lists, functions, and even classes**.
- **Example:**
- class Car:
- def __init__(self, brand, model):
- self.brand = brand
- self.model = model
-
- my_car = Car("Tesla", "Model S") # my_car is an object
- print(my_car.brand) # Output: Tesla

---

2. What is the difference between a class and an object?

- **Class**: A **blueprint** for creating objects.
- **Object**: An **instance** of a class with its own data.

Example:

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


dog1 = Dog("Buddy")  # Object (instance of class Dog)
dog2 = Dog("Charlie")
print(dog1.name)  # Output: Buddy
print(dog2.name)  # Output: Charlie
```

---

3. How do you define and create an object in Python?

- Define a class using class keyword.
- Create an object by **instantiating** the class.

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

p1 = Person("Alice", 25)  # Object creation
print(p1.name, p1.age)  # Output: Alice 25
```

---

4. What is the __init__ method in Python classes?

- The __init__ method is a **constructor** that initializes an object's attributes when it is created.

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

s1 = Student("John", "A")
print(s1.name, s1.grade)  # Output: John A
```

---

5. How do you access object attributes and methods?

- Use the **dot (.) operator** to access attributes and methods.

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

```
    def show_brand(self):
        return f"Car brand: {self.brand}"

car1 = Car("BMW")
print(car1.brand)        # Access attribute → Output: BMW
print(car1.show_brand()) # Access method → Output: Car brand: BMW
```

---

6. Can an object have multiple instances?

- **Yes**, multiple objects of the same class can exist, each with its own **unique attributes**.

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

laptop1 = Laptop("Apple", "MacBook Pro")
laptop2 = Laptop("Dell", "XPS 13")

print(laptop1.brand, laptop2.brand)  # Output: Apple Dell
```

---

7. What is the difference between instance variables and class variables?

- **Instance Variable**: Unique to each object.
- **Class Variable**: Shared among all instances.

```
class Employee:
    company = "Google"  # Class variable

    def __init__(self, name):
        self.name = name  # Instance variable
```

```
emp1 = Employee("Alice")
emp2 = Employee("Bob")

print(emp1.company, emp1.name)  # Google Alice
print(emp2.company, emp2.name)  # Google Bob
```

---

8. What is the self keyword in Python classes?

- **self** represents the **current instance** of the class.
- It is used to **access instance attributes and methods**.

```
class Animal:
    def __init__(self, species):
        self.species = species

    def show_species(self):
        return f"This is a {self.species}"

a = Animal("Dog")
print(a.show_species())  # Output: This is a Dog
```

---

9. What is the difference between object methods and static methods?

- **Instance Method (self)**: Operates on instance attributes.
- **Static Method (@staticmethod)**: Does not use self and works independently.

```
class MathOperations:
    @staticmethod
    def add(a, b):
        return a + b

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

---

10. How do you delete an object in Python?

- Use del to delete an object.

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

p = Person("John")
del p  # Deletes the object
```

Creating Classes

```
class Ns(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.course = ["NS", "Loni", "Vijayapur"]

    def speak(self):
        print("Hi, I am", self.name, "\nand my age is:", self.age, "\nUSN IS: ", self.usn)

    def change_age(self, age):
        self.age = age
```

```
    def add_usn(self, usn):
        self.usn = usn

loni1 = Ns("Nagaraj Loni", 21)
loni2 = Ns("Loni", 20)

loni1.change_age(20)
loni1.add_usn("3BR22CS***")

loni1.speak()

loni1.change_age(22)
loni1.add_usn("3BR22CS999")

# OUTPUT:
Hi, I am Nagaraj Loni
and my age is: 20
USN IS:  3BR22CS***
```

---

1. What is a class and an object in Python?

- Class: A blueprint for creating objects.
- Object: An instance of a class that holds data and behavior.

Example:

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

my_car = Car("Tesla", "Model S")  # Object
print(my_car.brand)  # Output: Tesla
```

---

2. How do you define a class in Python?

- Use the class keyword followed by a class name.
- Example:
- class Dog:
- def __init__(self, name):
- self.name = name
-
- d1 = Dog("Buddy")
- print(d1.name) # Output: Buddy

---

3. What is the __init__ method in Python classes?

- It is the constructor that initializes object attributes when an object is created.

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

s1 = Student("Alice", "A")
print(s1.name, s1.grade)  # Output: Alice A
```

---

4. What is the difference between instance variables and class variables?

- Instance Variables: Specific to an object, defined using self.
- Class Variables: Shared among all objects of a class.

```
class Employee:
    company = "Google"  # Class variable

    def __init__(self, name):
        self.name = name  # Instance variable
```

```
e1 = Employee("Alice")
e2 = Employee("Bob")

print(e1.company, e2.company)  # Output: Google Google
print(e1.name, e2.name)        # Output: Alice Bob
```

---

5. What is the purpose of the self keyword?

- self refers to the current instance of the class and allows access to instance attributes.

```
class Animal:
    def __init__(self, species):
        self.species = species

    def show_species(self):
        return f"This is a {self.species}"

a = Animal("Dog")
print(a.show_species())  # Output: This is a Dog
```

---

6. What is the difference between instance methods, class methods, and static methods?

- Instance Method (self): Works with instance attributes.
- Class Method (@classmethod): Works with class attributes.
- Static Method (@staticmethod): Does not use instance or class attributes.

```
class Example:
    class_variable = "Class Scope"

    def instance_method(self):
        return "Instance Method"

    @classmethod
```

```
    def class_method(cls):
        return f"Class Method: {cls.class_variable}"

    @staticmethod
    def static_method():
        return "Static Method"

obj = Example()
print(obj.instance_method())  # Output: Instance Method
print(Example.class_method()) # Output: Class Method: Class Scope
print(Example.static_method())# Output: Static Method
```

---

7. How do you create multiple objects from the same class?

- Instantiate the class multiple times.

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

laptop1 = Laptop("Apple", "MacBook Pro")
laptop2 = Laptop("Dell", "XPS 13")

print(laptop1.brand, laptop2.brand)  # Output: Apple Dell
```

---

8. How can you modify a class attribute from an object?

- Use self.class_variable inside a method or use ClassName.class_variable outside.

```
class Company:
    company_name = "Microsoft"  # Class variable

emp1 = Company()
```

```
emp1.company_name = "Google"  # Changes for emp1 only


print(emp1.company_name)  # Output: Google
print(Company.company_name)  # Output: Microsoft (unchanged)
```

---

9. How do you delete an object in Python?

- Use the del keyword.

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


p = Person("John")
del p  # Deletes the object
```

---

10. Can you create a class without the __init__ method?

- Yes, but you must manually set attributes after object creation.

```
class Book:
    pass


b1 = Book()
b1.title = "Python Basics"
print(b1.title)  # Output: Python Basics
```

Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

---

1. What is inheritance in Python?

Answer:
Inheritance is a feature in object-oriented programming where a child class inherits attributes and methods from a parent class. It allows code reuse and hierarchical classification.

Example:

```
class Parent:
    def show(self):
        print("This is the parent class")


class Child(Parent):
    pass  # Inherits everything from Parent


c = Child()
c.show()  # Output: This is the parent class
```

---

2. What are the different types of inheritance in Python?

Answer:
Python supports five types of inheritance:

1. Single Inheritance (One parent, one child)
2. Multiple Inheritance (One child, multiple parents)
3. Multilevel Inheritance (Parent → Child → Grandchild)
4. Hierarchical Inheritance (One parent, multiple children)
5. Hybrid Inheritance (Combination of multiple types)

Example of Multiple Inheritance:

```
class A:
    def method_A(self):
        print("Method from class A")

class B:
    def method_B(self):
        print("Method from class B")

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

c = C()
c.method_A()  # Output: Method from class A
c.method_B()  # Output: Method from class B
```

---

3. How does Python support multiple inheritance, and how is method resolution order (MRO) determined?

Answer:
Python uses the C3 Linearization (MRO) algorithm to determine method resolution in multiple inheritance. It follows the depth-first left-right approach while avoiding duplicate calls.

Use ClassName.__mro__ or help(ClassName) to check the order.

Example:

```
class A:
    def show(self):
        print("Class A")

class B(A):
    def show(self):
        print("Class B")
```

```
class C(A):
    def show(self):
        print("Class C")

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

d = D()
d.show()  # Output: Class B (MRO: D → B → C → A)
print(D.__mro__)  # Prints the Method Resolution Order
```

---

4. What is the difference between super() and direct parent method calling?

Answer:

- super() calls the parent class method dynamically, useful in multiple inheritance.
- Direct calling (Parent.method(self)) hardcodes the method, which can cause issues in inheritance chains.

Example using super():

```
class Parent:
    def greet(self):
        print("Hello from Parent")

class Child(Parent):
    def greet(self):
        super().greet()  # Calls Parent's method
        print("Hello from Child")

c = Child()
c.greet()
# Output:
```

```
# Hello from Parent
# Hello from Child
```

---

5. What is multilevel inheritance? Provide an example.

Answer:
Multilevel inheritance is when a child class inherits from another child class.

Example:

```
class Grandparent:
    def grandparent_method(self):
        print("Grandparent method")

class Parent(Grandparent):
    def parent_method(self):
        print("Parent method")

class Child(Parent):
    def child_method(self):
        print("Child method")

c = Child()
c.grandparent_method()  # Inherited from Grandparent
c.parent_method()       # Inherited from Parent
c.child_method()        # Own method
```

---

6. Can a child class override a parent class method?

Answer:
Yes, method overriding allows a child class to redefine a method inherited from the parent class.

Example:

```
class Parent:
    def show(self):
        print("Parent class method")

class Child(Parent):
    def show(self):  # Overriding Parent's method
        print("Child class method")

c = Child()
c.show()  # Output: Child class method
```

---

7. What is the isinstance() function, and how is it used in inheritance?

Answer:
isinstance(obj, Class) checks if an object is an instance of a class (or its subclasses).

Example:

```
class Animal:
    pass

class Dog(Animal):
    pass

d = Dog()
print(isinstance(d, Dog))    # True
print(isinstance(d, Animal)) # True (since Dog inherits from Animal)
print(isinstance(d, object)) # True (all classes inherit from `object`)
```

---

8. What is the issubclass() function in Python?

Answer:
issubclass(Class1, Class2) checks if Class1 is derived from Class2.

Example:

```
class Vehicle:
    pass

class Car(Vehicle):
    pass

print(issubclass(Car, Vehicle))  # True
print(issubclass(Vehicle, Car))  # False
```

---

9. What is hierarchical inheritance? Provide an example.

Answer:
Hierarchical inheritance is when multiple child classes inherit from the same parent class.

Example:

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

class Cat(Animal):
    def speak(self):
        print("Cat meows")

d = Dog()
c = Cat()
d.speak()  # Output: Dog barks
c.speak()  # Output: Cat meows
```

---

10. What is hybrid inheritance? How can it cause issues?

Answer:
Hybrid inheritance is a combination of multiple inheritance types, which can lead to method resolution issues.

Example:

```
class A:
    def show(self):
        print("A")


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


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


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


d = D()
d.show()  # Output: B (because of MRO: D → B → C → A)
```

To resolve conflicts, Python follows MRO.

Bonus: Can you prevent inheritance in Python?

Yes! You can prevent inheritance using final classes by raising an exception inside __init__ or using metaclasses.

Example:

```
class FinalClass:
    def __init_subclass__(cls, **kwargs):
        raise TypeError("Inheritance is not allowed")

class SubClass(FinalClass):  # This will raise an error
    pass
```

Overloading Methods

---

1. What is method overloading in Python?

Answer:
Method overloading allows multiple methods with the same name but different arguments. However, Python does not support true method overloading like Java or C++. Instead, it uses default arguments or *args and **kwargs to achieve similar functionality.

Example using default arguments:

```
class MathOperations:
    def add(self, a, b=0, c=0):
        return a + b + c


obj = MathOperations()
print(ob. Add(5))       # Output: 5
print(obj.add(5, 10))   # Output: 15
print(obj.add(5, 10, 20))  # Output: 35
```

---

2. How can you achieve method overloading in Python?

Answer:
Since Python doesn't support method overloading directly, we can use:

1. Default arguments
2. Variable-length arguments (*args, **kwargs)
3. Function overloading using @singledispatch decorator

Example using *args:

```
class Calculator:
    def multiply(self, *args):
        result = 1
        for num in args:
            result *= num
        return result
```

```
obj = Calculator()
print(obj.multiply(2, 3))       # Output: 6
print(obj.multiply(2, 3, 4))    # Output: 24
```

---

3. How does @singledispatch help in method overloading?

Answer:
The functools.singledispatch decorator enables function overloading based on argument type.

Example:

```
from functools import singledispatch

@singledispatch
def display(data):
    print("Default:", data)

@display.register(int)
def _(data):
    print("Integer:", data)

@display.register(str)
def _(data):
    print("String:", data)

display(10)    # Output: Integer: 10
display("Hi")  # Output: String: Hi
display(3.14)  # Output: Default: 3.14
```

---

4. Can you overload constructors (__init__) in Python?

Answer:
Python does not support multiple __init__ methods. However, we can use default arguments or *args to handle different scenarios.

Example:

```
class Person:
    def __init__(self, name, age=None):
        self.name = name
        self.age = age if age else "Not specified"


p1 = Person("Alice")
p2 = Person("Bob", 25)


print(p1.name, p1.age)  # Output: Alice Not specified
print(p2.name, p2.age)  # Output: Bob 25
```

---

5. What is the difference between method overloading and method overriding?

Answer:

Feature	Method Overloading	Method Overriding
Definition	Multiple methods with same name, different parameters in the same class	Redefining a method in the child class that exists in the parent class
Support in Python	Not directly supported	Supported
Achieved using	Default arguments, *args, @singledispatch	super() to call parent method
Example	add(self, a, b=0, c=0)	def show(self): super().show()

---

6. What happens if you define multiple methods with the same name in Python?

Answer:
Only the last-defined method will be used, as Python does not support multiple methods with the same name in a class.

Example:

```
class Example:
    def show(self):
        print("First version")

    def show(self):  # This overrides the previous method
        print("Second version")

obj = Example()
obj.show()  # Output: Second version
```

---

7. How can you overload an operator in Python?

Answer:
Operator overloading is done using special methods (dunder methods) like __add__, __sub__, etc.

Example of overloading + operator:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(2, 3)
v2 = Vector(4, 5)
result = v1 + v2  # Uses __add__ method
print(result.x, result.y)  # Output: 6 8
```

---

8. How does Python handle function overloading with type hints?

Answer:
Python's @overload from typing allows defining multiple function signatures for better type hints.

Example:

```
from typing import overload


class MathOperations:
    @overload
    def add(self, a: int, b: int) -> int: ...


    @overload
    def add(self, a: float, b: float) -> float: ...


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


m = MathOperations()
print(m.add(2, 3))      # Output: 5
print(m.add(2.5, 3.5))  # Output: 6.0
```

---

9. How do you overload a method based on the number of arguments?

Answer:
Using default arguments or *args.

Example:

```
class Example:
    def display(self, a=None, b=None):
        if a and b:
            print(f"Two values: {a}, {b}")
```

```
    elif a:
        print(f"One value: {a}")
    else:
        print("No value provided")

e = Example()
e.display()       # Output: No value provided
e.display(10)     # Output: One value: 10
e.display(10, 20) # Output: Two values: 10, 20
```

---

10. What are the limitations of method overloading in Python?

Answer:

1. Python does not support traditional overloading (i.e., multiple methods with the same name but different parameters).
2. The last defined method overwrites previous ones if they have the same name.
3. It must be simulated using *args, **kwargs, @singledispatch, or default arguments.
4. Overloading based on return types is not supported like in Java.

Bonus: Can Python support true method overloading in the future?

Python's design philosophy prioritizes simplicity and explicitness over method overloading. It is unlikely to be introduced natively, but decorators like @singledispatch provide limited overloading functionality.

Static Methods and Class Methods

---

1. What is the difference between a static method and a class method in Python?

Answer:

Feature	Static Method (@staticmethod)	Class Method (@classmethod)
Bound to	Class (but doesn't use self or cls)	Class (uses cls parameter)
Can modify class attributes?	No	Yes
Can access instance attributes?	No	No
Can access class attributes?	No	Yes
How to define?	@staticmethod decorator	@classmethod decorator
When to use?	Utility/helper functions that don't modify the class	Factory methods or when modifying class attributes

Example:

class Example:

```
    class_var = "Class Variable"

    @staticmethod
    def static_method():
        return "I am a static method"

    @classmethod
    def class_method(cls):
        return f"I am a class method. Accessing: {cls.class_var}"

print(Example.static_method())  # Output: I am a static method
print(Example.class_method())   # Output: I am a class method. Accessing: Class Variable
```

---

2. How do you define a static method in Python?

Answer:
A static method is defined using the @staticmethod decorator. It does not receive self or cls as its first argument.

Example:

```
class Utility:
    @staticmethod
    def add(a, b):
        return a + b


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

---

3. How do you define a class method in Python?

Answer:
A class method is defined using the @classmethod decorator. It must take cls as the first parameter, which represents the class itself.

Example:

```
class Example:
    class_var = "Hello"

    @classmethod
    def update_class_var(cls, new_value):
        cls.class_var = new_value


Example.update_class_var("New Value")
print(Example.class_var)  # Output: New Value
```

---

4. Can static methods access class variables? Why or why not?

Answer:
No, static methods cannot access class variables directly because they do not receive the cls parameter.

Example:

```
class Example:
    class_var = "Class Variable"

    @staticmethod
    def static_method():
        return Example.class_var  # Can access manually

print(Example.static_method())  # Output: Class Variable
```

To access class variables, you must manually reference the class name.

---

5. Can class methods modify instance variables? Why or why not?

Answer:
No, class methods cannot directly modify instance variables because they operate at the class level (using cls instead of self). However, they can modify class-level attributes.

Example:

```
class Example:
    class_var = 10

    @classmethod
    def modify_class_var(cls, value):
        cls.class_var = value

obj1 = Example()
obj2 = Example()

Example.modify_class_var(20)
print(obj1.class_var)  # Output: 20
```

```
print(obj2.class_var)  # Output: 20
```

---

6. When should you use a class method instead of a static method?

Answer:
Use a class method when:

- You need to modify class attributes.
- You want to implement a factory method (a method that returns instances of the class).

Use a static method when:

- The method does not need access to class or instance variables.
- You are writing utility/helper functions related to the class.

Example of a factory method:

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

    @classmethod
    def from_birth_year(cls, name, birth_year):
        return cls(name, 2025 - birth_year)  # Creating instance

p = Person.from_birth_year("Alice", 2000)
print(p.name, p.age)  # Output: Alice 25
```

---

7. Can you override a static method in Python?

Answer:
Yes, a static method can be overridden in a subclass.

Example:

```
class Parent:
    @staticmethod
    def show():
        return "Parent static method"

class Child(Parent):
    @staticmethod
    def show():
        return "Child static method"

print(Child.show())  # Output: Child static method
```

Since static methods are bound to the class, overriding replaces the method in the child class.

---

8. What happens if you call a static method using an instance?

Answer:
A static method can be called both from the class and an instance, but it does not have access to instance attributes.

Example:

```
class Example:
    @staticmethod
    def greet():
        return "Hello, static method!"

obj = Example()
print(obj.greet())   # Output: Hello, static method!
print(Example.greet())  # Output: Hello, static method!
```

Even when called from an instance, greet() does not receive self or cls.

---

9. How are static methods different from instance methods?

Answer:

Feature	Static Method (@staticmethod)	Instance Method
Requires self?	❌ No	✅ Yes
Can modify instance variables?	❌ No	✅ Yes
Can access class variables?	❌ No (unless manually referenced)	✅ Yes
Can be called on class?	✅ Yes	❌ No (without an instance)
Used for?	Utility/helper functions	Object-specific behavior

Example:

```
class Example:
    def instance_method(self):
        return "I am an instance method"

    @staticmethod
    def static_method():
        return "I am a static method"

obj = Example()
print(obj.instance_method())  # ✅ Needs an instance
print(Example.static_method())  # ✅ Can be called directly
```

---

10. Can a static method call another static method inside the same class?

Answer:
Yes, a static method can call another static method inside the same class by referencing it using ClassName.method_name().

Example:

```
class MathOperations:
    @staticmethod
    def add(a, b):
        return a + b

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

    @staticmethod
    def combined_operation(a, b):
        return MathOperations.add(a, b) * MathOperations.multiply(a, b)

print(MathOperations.combined_operation(2, 3))  # Output: (2+3) * (2*3) = 5 * 6 = 30
```

Bonus: What is a real-world use case of class methods?

A factory method using @classmethod allows creating different objects with predefined settings.

Example:

```
class Logger:
    def __init__(self, level):
        self.level = level

    @classmethod
    def debug_logger(cls):
        return cls("DEBUG")
```

```
    @classmethod
    def error_logger(cls):
        return cls("ERROR")

log1 = Logger.debug_logger()
log2 = Logger.error_logger()

print(log1.level)  # Output: DEBUG
print(log2.level)  # Output: ERROR
```

Private and Public Classes

---

1. What is the difference between public and private attributes in Python classes?

Answer:

- Public attributes can be accessed directly from outside the class.
- Private attributes (prefixed with _ or __) cannot be accessed directly outside the class.

Example:

```
class Example:
    def __init__(self):
        self.public_var = "I am public"
        self.__private_var = "I am private"

obj = Example()
print(obj.public_var)   # ✅ Allowed
print(obj.__private_var) # ❌ AttributeError
```

---

2. How do you define a private variable in Python?

Answer:
A private variable is defined using double underscores (__) before the variable name.

Example:

```
class Example:
    def __init__(self):
        self.__private_var = "I am private"

obj = Example()
print(obj.__private_var)  # ❌ AttributeError
```

---

3. Can you access a private variable outside a class? If yes, how?

Answer:
Yes, using name mangling (_ClassName__variable).

Example:

```
class Example:
    def __init__(self):
        self.__private_var = "I am private"


obj = Example()
print(obj._Example__private_var)  # ✅ Access using name mangling
```

---

4. What is name mangling in Python?

Answer:
Name mangling is the process where Python renames private attributes to _ClassName__variable to prevent accidental access.

Example:

```
class Example:
    def __init__(self):
        self.__private_var = "I am private"


print(dir(Example()))  # Lists `_Example__private_var`
```

---

5. Can you modify a private variable from outside the class?

Answer:
Not directly, but using name mangling, we can modify it.

Example:

```
class Example:
    def __init__(self):
        self.__private_var = "I am private"


obj = Example()
```

```
obj._Example__private_var = "Modified value"
print(obj._Example__private_var)  # ✅ Modified value
```

---

6. How do getter and setter methods help with private attributes?

Answer:
Getter and setter methods control access to private attributes.

Example:

```
class Example:
    def __init__(self):
        self.__private_var = "Private"

    def get_private_var(self):
        return self.__private_var

    def set_private_var(self, value):
        self.__private_var = value

obj = Example()
print(obj.get_private_var())  # ✅ Accessing private variable
obj.set_private_var("Updated Private")
print(obj.get_private_var())  # ✅ Updated value
```

---

7. What is the difference between a single underscore (_var) and a double underscore (__var) in Python?

Answer:

Prefix	Meaning
_var (Single underscore)	Conventionally protected, but still accessible
__var (Double underscore)	Private, uses name mangling

Example:

```
class Example:
```

```
    def __init__(self):
        self._protected = "Protected"
        self.__private = "Private"

obj = Example()
print(obj._protected)  # ✅ Accessible (not enforced)
print(obj.__private)   # ❌ AttributeError
```

---

8. Can a private method be accessed outside the class?

Answer:
Not directly, but using name mangling (_ClassName__method).

Example:

```
class Example:
    def __private_method(self):
        return "Private Method"

obj = Example()
print(obj._Example__private_method())  # ✅ Access using name mangling
```

---

9. When should you use private variables in Python?

Answer:
Use private variables when:

- You want to restrict access to class attributes.
- You want to prevent accidental modification of important data.
- You are implementing encapsulation to hide implementation details.

Example:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def get_balance(self):
```

```
        return self.__balance  # Controlled access


account = BankAccount(1000)
print(account.get_balance())  # ✅ Output: 1000
```

---

10. Can subclasses access private variables of a parent class?

Answer:
No, private variables are not inherited by child classes. However, they can still be accessed using name mangling.

Example:

```
class Parent:
    def __init__(self):
        self.__private_var = "Parent's Private"


class Child(Parent):
    def access_parent_private(self):
        return self._Parent__private_var  # Using name mangling


obj = Child()
print(obj.access_parent_private())  # ✅ Output: Parent's Private
```

Intermediate Python

Optional Parameters

---

1. What are optional parameters in Python functions?

Answer:

Optional parameters in Python functions are parameters that have default values assigned. If the caller does not pass a value, the function uses the default value.

Example Code:

```
def greet(name="Guest"):
    return f"Hello, {name}!"


print(greet())        # Output: Hello, Guest!
print(greet("Alice")) # Output: Hello, Alice!
```

---

2. How do you define optional parameters in a function?

Answer:

You define optional parameters by assigning a default value to them in the function definition.

Example Code:

```
def power(base, exponent=2):
    return base ** exponent


print(power(3))    # Output: 9 (3^2)
print(power(3, 3)) # Output: 27 (3^3)
```

---

3. What happens if an optional parameter is defined before a required parameter?

Answer:

Python will throw a SyntaxError because optional parameters (having default values) must come after required parameters.

Example (Incorrect Code):

```
def example(a=10, b):
    return a + b  # ❌ SyntaxError: non-default argument follows default argument
```

Corrected Code:

```
def example(b, a=10):
    return a + b


print(example(5)) # Output: 15
```

---

4. Can you use None as a default value for an optional parameter? Why?

Answer:

Yes, using None as a default value is common when the actual default needs to be computed dynamically.

Example Code:

```
def add_to_list(item, lst=None):
    if lst is None:  # Initialize only if lst is not provided
        lst = []
    lst.append(item)
    return lst


print(add_to_list(1))       # Output: [1]
print(add_to_list(2))       # Output: [2] (new list is created)
print(add_to_list(3, [10])) # Output: [10, 3] (existing list is modified)
```

---

5. Can functions have multiple optional parameters?

Answer:

Yes, functions can have multiple optional parameters.

Example Code:

```
def print_info(name="Unknown", age=18, city="Delhi"):
    return f"Name: {name}, Age: {age}, City: {city}"


print(print_info())                 # Output: Name: Unknown, Age: 18, City: Delhi
```

```
print(print_info("Alice", 25))          # Output: Name: Alice, Age: 25, City: Delhi
print(print_info("Bob", city="Mumbai")) # Output: Name: Bob, Age: 18, City: Mumbai
```

---

6. How do you use *args with optional parameters?

Answer:

You can mix *args with optional parameters, but optional parameters should come after *args.

Example Code:

```
def sum_numbers(*nums, multiplier=1):
    return sum(nums) * multiplier


print(sum_numbers(1, 2, 3))        # Output: 6 (multiplier = 1)
print(sum_numbers(1, 2, 3, multiplier=2)) # Output: 12 (multiplier = 2)
```

---

7. How do you use **kwargs with optional parameters?

Answer:

**kwargs allows passing optional keyword arguments dynamically.

Example Code:

```
def student_info(name, **details):
    return f"Name: {name}, Details: {details}"


print(student_info("Alice", age=20, city="NY"))
# Output: Name: Alice, Details: {'age': 20, 'city': 'NY'}
```

---

8. What is the difference between positional and keyword optional parameters?

Answer:

- Positional optional parameters rely on the order of arguments.
- Keyword optional parameters allow passing arguments with their names.

Example Code:

```
def order(item, quantity=1, price=10):
    return f"Item: {item}, Quantity: {quantity}, Total Price: {quantity * price}"
```

print(order("Book")) # Positional: Defaults used -> Output: Item: Book, Quantity: 1, Total Price: 10

print(order("Pen", price=5)) # Keyword: Override price -> Output: Item: Pen, Quantity: 1, Total Price: 5

print(order("Pencil", 3, 2)) # Positional: Override both -> Output: Item: Pencil, Quantity: 3, Total Price: 6

---

9. How do you use a function with both *args and **kwargs along with optional parameters?

Answer:

You can use *args for variable-length positional arguments and **kwargs for variable-length keyword arguments.

Example Code:

```
def complete_info(name, *hobbies, age=18, **details):
    return f"Name: {name}, Age: {age}, Hobbies: {hobbies}, Details: {details}"


print(complete_info("Alice", "Reading", "Traveling", city="NY", country="USA"))
```

Output: Name: Alice, Age: 18, Hobbies: ('Reading', 'Traveling'), Details: {'city': 'NY', 'country': 'USA'}

---

10. What is the impact of mutable default parameters in Python?

Answer:

Using mutable default values (e.g., lists, dictionaries) can lead to unexpected behavior because they persist across function calls.

Example Code (Problem):

```
def append_to_list(item, lst=[]):
    lst.append(item)
    return lst


print(append_to_list(1)) # Output: [1]
print(append_to_list(2)) # Output: [1, 2] (Unexpected!)
print(append_to_list(3)) # Output: [1, 2, 3] (Unexpected!)
```

Corrected Code (Using None as Default):

```
def append_to_list(item, lst=None):
    if lst is None:
        lst = []
    lst.append(item)
    return lst

print(append_to_list(1)) # Output: [1]
print(append_to_list(2)) # Output: [2] (Now works correctly)
```

Summary of Key Takeaways

1. Optional parameters have default values.
2. Defaults should come after required parameters in function definitions.
3. Use None instead of mutable types to avoid unexpected behavior.
4. You can mix required, optional, *args, and **kwargs for flexible function definitions.
5. Keyword arguments (**kwargs) allow handling dynamic optional parameters.

Static and Class Methods

---

1. What is the difference between a static method and a class method?

Answer:

- A static method (@staticmethod) belongs to the class but does not access or modify class attributes.
- A class method (@classmethod) takes the class (cls) as its first parameter and can modify class attributes.

Example Code:

```
class Example:
    class_var = "Class Variable"
```

```
    @staticmethod
    def static_method():
        return "I am a static method"

    @classmethod
    def class_method(cls):
        return f"I am a class method accessing: {cls.class_var}"

print(Example.static_method())  # Output: I am a static method
print(Example.class_method())   # Output: I am a class method accessing: Class Variable
```

---

2. When should you use a static method instead of a class method?

Answer:

- Use static methods when you don't need to access or modify class attributes.
- Use class methods when you need to modify or interact with class variables.

Example Code:

```
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y  # No class attribute access, so it's a static method

    @classmethod
    def create_instance(cls):
        return cls()  # Creates an instance, so it's a class method
```

---

3. Can a static method access instance variables?

Answer:

No, a static method cannot access instance variables because it does not receive self as a parameter.

Example Code:

```
class Test:
    def __init__(self, value):
        self.value = value

    @staticmethod
    def static_method():
        # return self.value  # ❌ Error: No access to instance variables
        return "Static method cannot access instance attributes"

obj = Test(10)
print(obj.static_method())  # Output: Static method cannot access instance attributes
```

---

4. Can a class method modify instance variables?

Answer:

No, a class method can modify class variables but not instance variables unless explicitly passed an instance.

Example Code:

```
class Test:
    class_var = 0  # Class variable

    def __init__(self, value):
        self.instance_var = value  # Instance variable

    @classmethod
    def modify_class_var(cls, new_value):
        cls.class_var = new_value  # ☑ Allowed

obj = Test(10)
Test.modify_class_var(100)
print(Test.class_var)  # Output: 100
```

---

5. What happens if you try to access self in a static method?

Answer:

You will get a NameError because self is not passed to static methods.

Example Code:

```
class Demo:
    def __init__(self, value):
        self.value = value

    @staticmethod
    def static_method():
        # print(self.value)  # ❌ NameError: name 'self' is not defined
        return "Static methods cannot use self"
```

---

6. Can a static method call another static method in the same class?

Answer:

Yes, a static method can call other static methods using the class name.

Example Code:

```
class Utils:
    @staticmethod
    def square(n):
        return n * n

    @staticmethod
    def cube(n):
        return n * Utils.square(n)  # Calling another static method

print(Utils.cube(3))  # Output: 27
```

---

7. Can a class method call a static method?

Answer:

Yes, a class method can call a static method using the class name.

Example Code:

```
class Operations:
    @staticmethod
    def multiply(x, y):
        return x * y

    @classmethod
    def multiply_by_two(cls, num):
        return cls.multiply(num, 2)  # Calling static method from class method

print(Operations.multiply_by_two(5))  # Output: 10
```

---

8. What happens if you override a static method in a subclass?

Answer:

Static methods can be overridden, but they do not get automatically bound to the subclass.

Example Code:

```
class Parent:
    @staticmethod
    def display():
        return "Static method in Parent"

class Child(Parent):
    @staticmethod
    def display():
        return "Static method in Child"

print(Parent.display())  # Output: Static method in Parent
print(Child.display())   # Output: Static method in Child
```

---

9. What happens if you override a class method in a subclass?

Answer:

A class method can be overridden in a subclass and will receive the subclass (cls) when called from the subclass.

Example Code:

```
class Parent:
    class_var = "Parent Variable"

    @classmethod
    def show(cls):
        return f"Class method in {cls.__name__}, Variable: {cls.class_var}"

class Child(Parent):
    class_var = "Child Variable"

print(Parent.show())  # Output: Class method in Parent, Variable: Parent Variable
print(Child.show())   # Output: Class method in Child, Variable: Child Variable
```

---

10. How do static and class methods behave with inheritance?

Answer:

- Static methods are inherited as is.
- Class methods are inherited and receive the subclass (cls) when called from a subclass.

Example Code:

```
class Parent:
    @staticmethod
    def static_method():
        return "Parent Static Method"

    @classmethod
    def class_method(cls):
```

```
        return f"Class method in {cls.__name__}"

class Child(Parent):
    pass

print(Child.static_method())  # Output: Parent Static Method (inherited without modification)
print(Child.class_method())   # Output: Class method in Child (receives Child as cls)
```

Key Takeaways

Feature	Static Method (@staticmethod)	Class Method (@classmethod)
Receives self?	❌ No	❌ No
Receives cls?	❌ No	☑ Yes
Access instance variables?	❌ No	❌ No
Access class variables?	❌ No	☑ Yes
Modify class variables?	❌ No	☑ Yes
Calls another static method?	☑ Yes	☑ Yes
Calls another class method?	❌ No	☑ Yes

Final Thoughts

- Use @staticmethod when the method does not need class or instance data.
- Use @classmethod when you need to modify class attributes or create instances dynamically.

Map Function

---

1. What is the map() function in Python?

Answer:

The map() function applies a given function to all items in an iterable (e.g., list, tuple) and returns a map object (which is an iterator).

Example Code:

```
def square(n):
    return n * n


numbers = [1, 2, 3, 4]
squared_numbers = map(square, numbers)
print(list(squared_numbers))  # Output: [1, 4, 9, 16]
```

---

2. What is the syntax of the map() function?

Answer:

The syntax of map() is:

```
map(function, iterable)
```

- function: The function to apply.
- iterable: The iterable whose elements will be processed.

Example Code:

```
map(str.upper, ["apple", "banana", "cherry"])  # Converts all strings to uppercase
```

---

3. Can map() be used with multiple iterables?

Answer:

Yes, map() can take multiple iterables. The function must accept the same number of arguments as there are iterables.

Example Code:

```
def add(x, y):
```

```
    return x + y

a = [1, 2, 3]
b = [4, 5, 6]
result = map(add, a, b)
print(list(result))  # Output: [5, 7, 9]
```

---

4. What is the difference between map() and a list comprehension?

Answer:

- map(): Uses a function and returns an iterator (lazy evaluation).
- List comprehension: Directly creates a new list.

Example Code:

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

# Using map()
print(list(map(lambda x: x * x, nums)))  # Output: [1, 4, 9, 16]

# Using list comprehension
print([x * x for x in nums])  # Output: [1, 4, 9, 16]
```

---

5. Can you use map() with a lambda function?

Answer:

Yes, lambda functions are commonly used with map().

Example Code:

```
numbers = [1, 2, 3, 4]
squared = map(lambda x: x ** 2, numbers)
print(list(squared))  # Output: [1, 4, 9, 16]
```

---

6. What does map() return? How do you get the values?

Answer:

map() returns a map object (iterator). You can convert it to a list, tuple, or set.

Example Code:

```
nums = [1, 2, 3]
doubled = map(lambda x: x * 2, nums)

print(type(doubled))  # Output: <class 'map'>
print(list(doubled))  # Output: [2, 4, 6]
```

---

7. How can you apply map() to a dictionary?

Answer:

You can use map() to transform dictionary values.

Example Code:

```
prices = {"apple": 100, "banana": 50, "cherry": 75}
discounted_prices = dict(map(lambda item: (item[0], item[1] * 0.9), prices.items()))
print(discounted_prices)
# Output: {'apple': 90.0, 'banana': 45.0, 'cherry': 67.5}
```

---

8. What happens if the iterables passed to map() have different lengths?

Answer:

If the iterables have different lengths, map() stops at the shortest one.

Example Code:

```
a = [1, 2, 3]
b = [4, 5]

result = map(lambda x, y: x + y, a, b)
print(list(result))  # Output: [5, 7] (Stops at the shortest iterable)
```

---

9. How do you use map() with str.split() and str.strip()?

Answer:

You can use map() to apply string methods to multiple strings.

Example Code:

```
words = ["  hello ", " world ", " python  "]
cleaned_words = list(map(str.strip, words))
print(cleaned_words)  # Output: ['hello', 'world', 'python']
```

---

10. How do you filter None values from map() results?

Answer:

Use filter() along with map().

Example Code:

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

def even_square(n):
    return n ** 2 if n % 2 == 0 else None

result = filter(None, map(even_square, nums))
print(list(result))  # Output: [4, 16] (Filters out None values)
```

Key Takeaways

1. map() applies a function to all elements in an iterable.
2. Returns a lazy iterator, so use list() to get results.
3. Supports multiple iterables if the function takes multiple arguments.
4. Stops at the shortest iterable when multiple iterables are passed.
5. Can be used with lambda functions for concise code.

Filter Function

---

1. What is the filter() function in Python?

Answer:

The filter() function applies a given function to all elements in an iterable and returns only those elements for which the function returns True.

Syntax:

filter(function, iterable)

- function: A function that returns True or False.
- iterable: The sequence to filter.

Example Code:

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


def is_even(n):
    return n % 2 == 0


even_numbers = filter(is_even, numbers)
print(list(even_numbers))  # Output: [2, 4]
```

---

2. How does filter() differ from map() in Python?

Answer:

- map(function, iterable) applies a function to all elements and returns the transformed values.
- filter(function, iterable) applies a function and returns only elements where the function evaluates to True.

Example Code:

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


# map() applies function to all elements
print(list(map(lambda x: x * x, nums)))  # Output: [1, 4, 9, 16]
```

```
# filter() only keeps elements where function returns True
print(list(filter(lambda x: x % 2 == 0, nums)))  # Output: [2, 4]
```

---

3. Can you use filter() with a lambda function?

Answer:

Yes, lambda functions are commonly used with filter().

Example Code:

```
numbers = [5, 10, 15, 20, 25, 30]
filtered = filter(lambda x: x > 15, numbers)
print(list(filtered))  # Output: [20, 25, 30]
```

---

4. What does filter(None, iterable) do?

Answer:

Using None as the function in filter() removes all falsy values (None, 0, False, "", [], {}).

Example Code:

```
data = [0, 1, False, "Hello", "", [], None, 42]
filtered = filter(None, data)
print(list(filtered))  # Output: [1, 'Hello', 42]
```

---

5. Can filter() be used with multiple iterables?

Answer:

No, filter() only works on one iterable at a time. If multiple iterables are needed, use zip().

Example Code:

```
nums = [10, 20, 30, 40]
thresholds = [15, 25, 35, 45]

result = filter(lambda pair: pair[0] > pair[1], zip(nums, thresholds))
print(list(result))  # Output: []
```

(Each tuple (num, threshold) is compared.)

---

6. How can you filter a list of dictionaries?

Answer:

You can filter a list of dictionaries by applying a condition on dictionary values.

Example Code:

```
students = [
    {"name": "Alice", "grade": 85},
    {"name": "Bob", "grade": 70},
    {"name": "Charlie", "grade": 90}
]


passed_students = filter(lambda s: s["grade"] >= 80, students)
print(list(passed_students))
# Output: [{'name': 'Alice', 'grade': 85}, {'name': 'Charlie', 'grade': 90}]
```

---

7. How do you filter even numbers from a list?

Answer:

Use filter() with a function that checks for even numbers.

Example Code:

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # Output: [2, 4, 6]
```

---

8. How do you filter words based on length from a list?

Answer:

Use filter() to check string length.

Example Code:

```
words = ["apple", "banana", "kiwi", "grape", "mango"]
short_words = filter(lambda word: len(word) <= 5, words)
print(list(short_words))  # Output: ['apple', 'kiwi', 'grape']
```

---

9. How do you filter negative numbers from a list?

Answer:

Use filter() with a condition that checks for negative numbers.

Example Code:

```
numbers = [-10, -5, 0, 5, 10]
positive_numbers = filter(lambda x: x >= 0, numbers)
print(list(positive_numbers))  # Output: [0, 5, 10]
```

---

10. How do you filter elements from a list that contain a specific substring?

Answer:

Use filter() with the in operator.

Example Code:

```
fruits = ["apple", "banana", "cherry", "grape"]
filtered_fruits = filter(lambda fruit: "a" in fruit, fruits)
print(list(filtered_fruits))  # Output: ['apple', 'banana', 'grape']
```

Key Takeaways

1. filter() keeps only elements where the function returns True.
2. Returns an iterator, so use list(), tuple(), or set() to get values.
3. filter(None, iterable) removes falsy values (0, None, False, etc.).
4. Works on a single iterable, but can be used with zip() for multiple iterables.
5. Useful for filtering numbers, strings, and dictionaries.

Lambda Function

---

1. What is a lambda function in Python?

Answer:

A lambda function in Python is an anonymous function (a function without a name). It is created using the lambda keyword and can have multiple arguments but only a single expression.

Syntax:

lambda arguments: expression

Example Code:

```
square = lambda x: x * x

print(square(5))  # Output: 25
```

---

2. How is a lambda function different from a normal function?

Answer:

- Lambda function: Anonymous, single expression, inline.
- Normal function: Named, can have multiple expressions and statements.

Example Code:

```
# Normal function

def add(a, b):

    return a + b


# Lambda function

add_lambda = lambda a, b: a + b


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

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

---

3. Can a lambda function have multiple arguments?

Answer:

Yes, lambda functions can have multiple arguments.

Example Code:

```
multiply = lambda x, y: x * y
print(multiply(3, 4))  # Output: 12
```

---

4. Can a lambda function return multiple values?

Answer:

No, a lambda function can return only one expression. However, you can return a tuple.

Example Code:

```
multiple_values = lambda x, y: (x + y, x * y)
print(multiple_values(3, 4))  # Output: (7, 12)
```

---

5. How do you use a lambda function with map()?

Answer:

The map() function applies a lambda function to each element in an iterable.

Example Code:

```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x ** 2, nums))
print(squared)  # Output: [1, 4, 9, 16]
```

---

6. How do you use a lambda function with filter()?

Answer:

The filter() function keeps only the elements where the lambda function returns True.

Example Code:

```
nums = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, nums))
print(even_numbers)  # Output: [2, 4, 6]
```

---

7. How do you use a lambda function with sorted()?

Answer:

You can use lambda as a custom sorting key.

Example Code:

```
students = [("Alice", 25), ("Bob", 20), ("Charlie", 23)]
students_sorted = sorted(students, key=lambda x: x[1])
print(students_sorted)  # Output: [('Bob', 20), ('Charlie', 23), ('Alice', 25)]
```

---

8. Can a lambda function use an if-else statement?

Answer:

Yes, but it must be a single-line expression.

Example Code:

```
max_value = lambda a, b: a if a > b else b
print(max_value(10, 20))  # Output: 20
```

---

9. How do you use a lambda function inside a dictionary?

Answer:

You can store lambda functions as dictionary values.

Example Code:

```
operations = {
    "add": lambda x, y: x + y,
    "subtract": lambda x, y: x - y,
    "multiply": lambda x, y: x * y
}

print(operations["add"](5, 3))       # Output: 8
print(operations["subtract"](10, 4))  # Output: 6
print(operations["multiply"](6, 7))   # Output: 42
```

---

10. Can a lambda function call another function?

Answer:

Yes, a lambda function can call other functions.

Example Code:

```
def square(n):
    return n * n


square_lambda = lambda x: square(x)


print(square_lambda(5))  # Output: 25
```

Key Takeaways

1. Lambda functions are anonymous (no name) and written in a single line.
2. Can take multiple arguments but only one expression.
3. Used with map(), filter(), sorted(), and dictionaries.
4. Supports if-else statements in a single line.
5. Can call other functions inside them.

Introduction to Collections

---

1. What is the Counter class in the collections module?

Answer:

The Counter class is a subclass of dict used to count hashable objects.

Example Code:

```
from collections import Counter

words = ["apple", "banana", "apple", "orange", "banana", "apple"]
counter = Counter(words)
print(counter)  # Output: Counter({'apple': 3, 'banana': 2, 'orange': 1})

# Get the most common element
print(counter.most_common(1))  # Output: [('apple', 3)]
```

---

2. What is a deque and how is it different from a list?

Answer:

A deque (double-ended queue) is optimized for fast appends and pops from both ends, whereas a list is slow for operations at the start.

Example Code:

```
from collections import deque

dq = deque([1, 2, 3])
dq.append(4)      # Add to the right
dq.appendleft(0)  # Add to the left
print(dq)         # Output: deque([0, 1, 2, 3, 4])

dq.pop()          # Remove from the right
dq.popleft()      # Remove from the left
print(dq)         # Output: deque([1, 2, 3])
```

---

3. What is defaultdict and why is it useful?

Answer:

defaultdict is a subclass of dict that provides a default value for missing keys instead of raising a KeyError.

Example Code:

```
from collections import defaultdict


dd = defaultdict(int)  # Default value for missing keys is 0

dd['apple'] += 1

print(dd)  # Output: defaultdict(<class 'int'>, {'apple': 1})


# Example with list

dd_list = defaultdict(list)

dd_list["fruits"].append("apple")

print(dd_list)  # Output: defaultdict(<class 'list'>, {'fruits': ['apple']})
```

---

4. What is an OrderedDict and how does it differ from a regular dictionary?

Answer:

An OrderedDict remembers the insertion order of keys, unlike a regular dictionary (before Python 3.7).

Example Code:

```
from collections import OrderedDict


od = OrderedDict()

od["apple"] = 3

od["banana"] = 2

od["cherry"] = 5

print(od)  # Output: OrderedDict([('apple', 3), ('banana', 2), ('cherry', 5)])
```

Regular dictionary (since Python 3.7+) also maintains order, but OrderedDict has extra methods.

---

5. What is a namedtuple and why use it?

Answer:

A namedtuple provides a lightweight way to create immutable objects with named fields.

Example Code:

```
from collections import namedtuple

Person = namedtuple("Person", ["name", "age"])
p = Person(name="Alice", age=30)

print(p.name)  # Output: Alice
print(p.age)   # Output: 30
```

---

6. What is ChainMap and when is it useful?

Answer:

ChainMap groups multiple dictionaries together and treats them as a single dictionary.

Example Code:

```
from collections import ChainMap

dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}

cm = ChainMap(dict1, dict2)
print(cm["a"])  # Output: 1
print(cm["b"])  # Output: 2 (Takes from the first dictionary)
print(cm["c"])  # Output: 4
```

---

7. How do you get the n most common elements using Counter?

Answer:

Use the .most_common(n) method.

Example Code:

```
from collections import Counter

data = ['apple', 'banana', 'orange', 'apple', 'banana', 'apple']
counter = Counter(data)

print(counter.most_common(2))  # Output: [('apple', 3), ('banana', 2)]
```

---

8. How do you rotate a deque in Python?

Answer:

Use the .rotate(n) method.

Example Code:

```
from collections import deque

dq = deque([1, 2, 3, 4, 5])
dq.rotate(2)  # Rotates right by 2
print(dq)  # Output: deque([4, 5, 1, 2, 3])

dq.rotate(-1)  # Rotates left by 1
print(dq)  # Output: deque([5, 1, 2, 3, 4])
```

---

9. How can you merge two Counter objects?

Answer:

Use the + operator or update().

Example Code:

```
from collections import Counter

c1 = Counter({'apple': 2, 'banana': 3})
c2 = Counter({'banana': 1, 'cherry': 5})
```

```
# Merge counters
merged = c1 + c2
print(merged)  # Output: Counter({'cherry': 5, 'banana': 4, 'apple': 2})
```

---

10. How do you use defaultdict to group elements?

Answer:

You can use defaultdict(list) to group elements by a common key.

Example Code:

```
from collections import defaultdict

data = [('fruit', 'apple'), ('fruit', 'banana'), ('vegetable', 'carrot'), ('fruit', 'grape')]

grouped = defaultdict(list)
for category, item in data:
    grouped[category].append(item)

print(grouped)  # Output: defaultdict(<class 'list'>, {'fruit': ['apple', 'banana', 'grape'], 'vegetable': ['carrot']})
```

Key Takeaways

1. Counter: Counts occurrences of elements.
2. deque: Optimized for fast appends/pops from both ends.
3. defaultdict: Provides default values for missing keys.
4. OrderedDict: Preserves insertion order (useful before Python 3.7).
5. namedtuple: Immutable tuple-like objects with named fields.
6. ChainMap: Groups multiple dictionaries into one.
7. most_common(n): Finds the most common elements in a Counter.
8. .rotate(n): Rotates a deque.
9. Merging Counter objects: Using + operator or .update().
10. Grouping elements with defaultdict(list): Efficient way to categorize data.

Named Tuple

---

1. What is namedtuple in Python?

Answer:

A namedtuple is a subclass of Python's built-in tuple that allows you to create immutable, lightweight objects with named fields, making the code more readable.

Example Code:

```
from collections import namedtuple

Person = namedtuple("Person", ["name", "age"])
p = Person(name="Alice", age=25)

print(p.name)  # Output: Alice
print(p.age)   # Output: 25
```

---

2. How is namedtuple different from a regular tuple?

Answer:

- Regular tuple: Access elements using indexing.
- Namedtuple: Access elements using dot notation or indexing.

Example Code:

```
from collections import namedtuple

# Regular tuple
person_tuple = ("Alice", 25)
print(person_tuple[0])  # Output: Alice

# Named tuple
Person = namedtuple("Person", ["name", "age"])
p = Person("Alice", 25)
print(p.name)  # Output: Alice
```

---

3. How do you assign default values to fields in namedtuple?

Answer:

Use _replace() or defaults via NamedTuple (Python 3.7+).

Example Code:

```
from collections import namedtuple

Person = namedtuple("Person", ["name", "age"])
p = Person("Alice", 25)

# Use _replace() to change a value
p = p._replace(age=30)
print(p)  # Output: Person(name='Alice', age=30)
```

For default values in Python 3.7+:

```
from typing import NamedTuple

class Person(NamedTuple):
    name: str
    age: int = 25

p = Person("Alice")
print(p)  # Output: Person(name='Alice', age=25)
```

---

4. How do you convert a namedtuple to a dictionary?

Answer:

Use the _asdict() method.

Example Code:

```
from collections import namedtuple

Person = namedtuple("Person", ["name", "age"])
```

```
p = Person("Alice", 25)

print(p._asdict())
# Output: {'name': 'Alice', 'age': 25}
```

---

5. Can a namedtuple be mutable?

Answer:

No, namedtuple is immutable. However, you can use _replace() to create a new instance with modified values.

Example Code:

```
from collections import namedtuple

Person = namedtuple("Person", ["name", "age"])
p = Person("Alice", 25)

# Trying to change value (will raise an error)
# p.age = 30  # AttributeError: can't set attribute

# Instead, use _replace()
p_new = p._replace(age=30)
print(p_new)  # Output: Person(name='Alice', age=30)
```

---

6. How do you add methods to a namedtuple?

Answer:

Use a class-based named tuple with NamedTuple (Python 3.6+).

Example Code:

```
from typing import NamedTuple

class Person(NamedTuple):
    name: str
    age: int
```

```
    def greet(self):
        return f"Hello, my name is {self.name} and I am {self.age} years old."

p = Person("Alice", 25)
print(p.greet())
# Output: Hello, my name is Alice and I am 25 years old.
```

---

7. How do you use namedtuple with *args and **kwargs?

Answer:

You can create a named tuple dynamically using *args and **kwargs.

Example Code:

```
from collections import namedtuple

Person = namedtuple("Person", ["name", "age"])

# Using *args
p1 = Person(*["Alice", 25])
print(p1)  # Output: Person(name='Alice', age=25)

# Using **kwargs
p2 = Person(**{"name": "Bob", "age": 30})
print(p2)  # Output: Person(name='Bob', age=30)
```

---

8. How do you check if a field exists in a namedtuple?

Answer:

Use _fields to check available fields.

Example Code:

```
from collections import namedtuple
```

```
Person = namedtuple("Person", ["name", "age"])
p = Person("Alice", 25)

print("name" in p._fields)  # Output: True
print("gender" in p._fields)  # Output: False
```

---

9. How do you create a namedtuple dynamically?

Answer:

Use namedtuple() with a string of field names.

Example Code:

```
from collections import namedtuple

# Creating a namedtuple dynamically
Employee = namedtuple("Employee", "name age department")
e = Employee("John", 30, "HR")

print(e)  # Output: Employee(name='John', age=30, department='HR')
```

---

10. How do you iterate over a namedtuple?

Answer:

You can iterate using indexing, _fields, or _asdict().

Example Code:

```
from collections import namedtuple

Person = namedtuple("Person", ["name", "age"])
p = Person("Alice", 25)

# Iterating using indexing
for value in p:
    print(value)
```

```
# Iterating using _fields
for field in p._fields:
    print(field, getattr(p, field))

# Iterating using _asdict()
for key, value in p._asdict().items():
    print(key, value)
```

Output:

```
Alice
25
name Alice
age 25
name Alice
age 25
```

Key Takeaways

1. namedtuple provides named fields, making tuples more readable.
2. Immutable: You can't modify fields directly.
3. Supports _replace(), _fields, _asdict(), _make() for manipulation.
4. Can be used dynamically via namedtuple("ClassName", "field1 field2").
5. Can be converted to a dictionary using _asdict().
6. Supports method extensions using NamedTuple (Python 3.6+).
7. Easier to iterate than regular tuples.

Deque

---

1. What is deque in Python, and why is it useful?

Answer:

deque (short for double-ended queue) is a data structure from the collections module that allows fast appends and pops from both ends, unlike a list, which is slow for such operations.

Example Code:

```
from collections import deque

dq = deque([1, 2, 3])
dq.append(4)       # Add to the right
dq.appendleft(0)   # Add to the left

print(dq)  # Output: deque([0, 1, 2, 3, 4])
```

---

2. How do you remove elements from both ends of a deque?

Answer:

Use .pop() to remove from the right and .popleft() to remove from the left.

Example Code:

```
from collections import deque

dq = deque([1, 2, 3, 4, 5])
dq.pop()      # Removes 5 from the right
dq.popleft()  # Removes 1 from the left

print(dq)  # Output: deque([2, 3, 4])
```

---

3. How do you rotate a deque?

Answer:

Use .rotate(n).

- A positive n rotates right.

- A negative n rotates left.

Example Code:

```
from collections import deque

dq = deque([1, 2, 3, 4, 5])
dq.rotate(2)  # Right rotation by 2
print(dq)  # Output: deque([4, 5, 1, 2, 3])

dq.rotate(-1)  # Left rotation by 1
print(dq)  # Output: deque([5, 1, 2, 3, 4])
```

---

4. How is deque better than a list?

Answer:

- deque has O(1) time complexity for append() and pop() at both ends.
- list has O(n) complexity for insert(0, x) and pop(0) (removing from the front).

Example Code (Performance Comparison):

```
from collections import deque
import time

# List performance
lst = []
start = time.time()
for i in range(10**6):
    lst.insert(0, i)  # O(n)
print("List time:", time.time() - start)

# Deque performance
dq = deque()
start = time.time()
for i in range(10**6):
```

```
    dq.appendleft(i)  # O(1)
print("Deque time:", time.time() - start)
```

- Result: deque is significantly faster for inserting/removing from the front.

---

5. How do you set a maximum size for a deque?

Answer:

Use deque(maxlen=N). When the limit is reached, older elements automatically get removed.

Example Code:

```
from collections import deque

dq = deque(maxlen=3)
dq.append(1)
dq.append(2)
dq.append(3)
print(dq)  # Output: deque([1, 2, 3], maxlen=3)

dq.append(4)  # 1 is removed automatically
print(dq)  # Output: deque([2, 3, 4], maxlen=3)
```

---

6. How do you extend a deque with multiple values?

Answer:

- Use .extend(iterable) to add elements to the right.
- Use .extendleft(iterable) to add elements to the left (order is reversed).

Example Code:

```
from collections import deque

dq = deque([3, 4])
dq.extend([5, 6])       # Adds [5,6] to the right
print(dq)  # Output: deque([3, 4, 5, 6])
```

```
dq.extendleft([2, 1])   # Adds [2,1] to the left (order reversed)
print(dq)  # Output: deque([1, 2, 3, 4, 5, 6])
```

---

7. How do you reverse a deque?

Answer:

Use .reverse() in-place or reversed() for an iterator.

Example Code:

```
from collections import deque

dq = deque([1, 2, 3, 4])
dq.reverse()
print(dq)  # Output: deque([4, 3, 2, 1])

# Alternative method
print(deque(reversed(dq)))  # Output: deque([1, 2, 3, 4])
```

---

8. How do you clear all elements in a deque?

Answer:

Use .clear() to remove all elements.

Example Code:

```
from collections import deque

dq = deque([1, 2, 3])
dq.clear()
print(dq)  # Output: deque([])
```

---

9. How do you count occurrences of an element in a deque?

Answer:

Use .count(x) to count occurrences.

Example Code:

```
from collections import deque

dq = deque([1, 2, 3, 2, 4, 2, 5])
print(dq.count(2))  # Output: 3
```

---

10. How do you use deque to implement a queue?

Answer:

A queue follows FIFO (First-In, First-Out). Use .append(x) to enqueue and .popleft() to dequeue.

Example Code (Queue Implementation):

```
from collections import deque

queue = deque()

# Enqueue
queue.append("A")
queue.append("B")
queue.append("C")

print(queue)  # Output: deque(['A', 'B', 'C'])

# Dequeue
print(queue.popleft())  # Output: A
print(queue)  # Output: deque(['B', 'C'])
```

Key Takeaways

- ☑ deque is faster than lists for inserting/removing at both ends.
- ☑ .append() / .appendleft() add elements to the right/left.
- ☑ .pop() / .popleft() remove elements from the right/left.
- ☑ .rotate(n) shifts elements right (positive n) or left (negative n).
- ☑ .extend(iterable) and .extendleft(iterable) add multiple elements.
- ☑ .count(x) counts occurrences of an element.

- ☑ deque(maxlen=N) auto-removes older elements when full.
- ☑ .reverse() reverses a deque in place.
- ☑ .clear() removes all elements.

📒 Advanced Python 📒

Overview of Python

---

1. What is Python, and what are its key features?

Answer:

Python is a high-level, interpreted, dynamically typed, and object-oriented programming language known for its simplicity and readability.

Key Features:

- Easy to Learn & Readable
- Dynamically Typed (no need to declare variable types)
- Interpreted (runs line by line)
- Object-Oriented & Functional Programming Support
- Extensive Libraries (NumPy, Pandas, TensorFlow, etc.)
- Cross-Platform (works on Windows, Mac, Linux)

Example Code:

```
print("Hello, Python!")  # Simple Python program
```

---

2. How is Python interpreted and dynamically typed?

Answer:

Python does not require explicit type declaration (dynamic typing), and the code is executed line by line (interpreted).

Example Code:

```
x = 10      # Integer
x = "Hello" # Now it's a string
print(x)    # Output: Hello
```

📌 In C/C++, you must declare types (int x = 10;), but in Python, variables can change types dynamically.

---

3. What are Python's built-in data types?

Answer:

Python provides several built-in types:

- Numeric: int, float, complex
- Boolean: True, False
- Sequence: list, tuple, range
- Text: str
- Set types: set, frozenset
- Mapping: dict
- Binary types: bytes, bytearray, memoryview

Example Code:

```
num = 10        # int
pi = 3.14       # float
is_python = True  # bool
text = "Python" # str
lst = [1, 2, 3] # list
tpl = (4, 5, 6) # tuple
st = {7, 8, 9}  # set
dct = {"a": 1, "b": 2}  # dict


print(type(lst))  # Output: <class 'list'>
```

---

4. What are lists, tuples, sets, and dictionaries in Python?

Answer:

Data Type	Ordered	Mutable	Allows Duplicates	Example
List	☑ Yes	☑ Yes	☑ Yes	[1, 2, 3]
Tuple	☑ Yes	✗ No	☑ Yes	(1, 2, 3)
Set	✗ No	☑ Yes	✗ No	{1, 2, 3}
Dict	☑ Yes	☑ Yes	✗ No (keys)	{"a": 1, "b": 2}

Example Code:

```
lst = [1, 2, 3]       # List (mutable)
tpl = (1, 2, 3)       # Tuple (immutable)
st = {1, 2, 3}        # Set (unique values)
dct = {"a": 1, "b": 2}  # Dictionary (key-value)

lst.append(4)  # Works
# tpl[0] = 10  # Error! Tuples are immutable

print(lst, tpl, st, dct)
```

---

5. What is the difference between is and == in Python?

Answer:

- == checks value equality.
- is checks memory address (identity) equality.

Example Code:

```
a = [1, 2, 3]
b = a  # Both point to the same object
c = [1, 2, 3]  # Different object with same value

print(a == c)  # True (values are equal)
print(a is c)  # False (different objects)
print(a is b)  # True (same memory reference)
```

---

6. What are Python functions, and how do you define them?

Answer:

Functions are reusable blocks of code defined using def.

Example Code:

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

```
print(greet("Alice"))  # Output: Hello, Alice!
```

---

7. What are lambda functions in Python?

Answer:

A lambda function is an anonymous function with a single line of expression.

Example Code:

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

---

8. What are Python's control flow statements?

Answer:

Python supports:

- Conditional Statements: if, elif, else
- Loops: for, while
- Loop Control: break, continue, pass

Example Code:

```
x = 10
if x > 5:
    print("Greater than 5")
else:
    print("Less or equal to 5")


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

---

9. What are Python modules and packages?

Answer:

- A module is a single Python file (.py) containing functions and variables.
- A package is a collection of modules inside a directory with an __init__.py file.

Example Code (Creating a Module math_utils.py):

```
# math_utils.py
def add(a, b):
    return a + b
```

Using the Module in Another Script:

```
import math_utils

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

---

10. How does Python handle memory management?

Answer:

Python uses:

- Automatic garbage collection (via reference counting and cyclic GC)
- Memory allocation using private heap

Example Code (Garbage Collection Demo):

```
import gc

class Test:
    def __del__(self):
        print("Object deleted")

obj = Test()
del obj  # Explicitly deleting the object
gc.collect()  # Forces garbage collection
```

📌 The __del__() method runs when an object is garbage-collected.

Key Takeaways

☑ Python is an interpreted, dynamically typed, and object-oriented language.
☑ is checks identity, while == checks value equality.
☑ Lists, Tuples, Sets, and Dicts serve different use cases.
☑ Lambda functions provide a quick way to define small functions.

☑ Modules & Packages organize code effectively.
☑ Garbage collection automatically manages memory.

Dunder/Magic Methods

---

1. What are Dunder (Magic) Methods in Python?

Answer:

Dunder (double underscore) or magic methods are special methods in Python classes that start and end with __ (double underscores). They allow operator overloading, object creation, and customization of built-in behavior.

Example Code:

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


obj = Demo("Python")
print(obj.__class__)  # Output: <class '__main__.Demo'>
```

---

2. What is __init__ method in Python?

Answer:

__init__ is a constructor in Python, called automatically when an object is instantiated.

Example Code:

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

```
p = Person("Alice", 25)
print(p.name, p.age)  # Output: Alice 25
```

---

3. What is __str__ and __repr__? How do they differ?

Answer:

- __str__ returns a user-friendly string representation of an object.
- __repr__ returns an unambiguous string representation, mainly for debugging.

Example Code:

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

    def __str__(self):
        return f"Person: {self.name}"  # User-friendly

    def __repr__(self):
        return f"Person('{self.name}')"  # Debugging-friendly

p = Person("Alice")
print(str(p))   # Output: Person: Alice
print(repr(p))  # Output: Person('Alice')
```

---

4. How does __len__ work in Python?

Answer:

__len__ defines the behavior of len() for an object.

Example Code:

```
class Container:
    def __init__(self, items):
        self.items = items
```

```
    def __len__(self):
        return len(self.items)

c = Container([1, 2, 3, 4])
print(len(c))  # Output: 4
```

---

5. How can we overload the + operator using __add__?

Answer:

__add__ allows us to define custom behavior for the + operator.

Example Code:

```
class Number:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return Number(self.value + other.value)

    def __str__(self):
        return str(self.value)

n1 = Number(10)
n2 = Number(20)
result = n1 + n2
print(result)  # Output: 30
```

---

6. What is __call__ method in Python?

Answer:

__call__ allows an object to be called like a function.

Example Code:

```
class Multiplier:
```

```
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, number):
        return number * self.factor

double = Multiplier(2)
print(double(5))  # Output: 10
```

---

7. What is __getitem__ and __setitem__ in Python?

Answer:

- __getitem__ allows index-based access like lists.
- __setitem__ allows modification of elements by index.

Example Code:

```
class CustomList:
    def __init__(self):
        self.data = {}

    def __getitem__(self, index):
        return self.data.get(index, "Not Found")

    def __setitem__(self, index, value):
        self.data[index] = value

c = CustomList()
c[0] = "Hello"
print(c[0])  # Output: Hello
print(c[1])  # Output: Not Found
```

---

8. How does __eq__ work for comparing objects?

Answer:

__eq__ allows us to define custom behavior for == operator.

Example Code:

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

    def __eq__(self, other):
        return self.name == other.name and self.age == other.age

p1 = Person("Alice", 30)
p2 = Person("Alice", 30)
p3 = Person("Bob", 25)

print(p1 == p2)  # Output: True
print(p1 == p3)  # Output: False
```

---

9. How do __enter__ and __exit__ work in Python?

Answer:

These methods allow objects to be used with with statements for resource management.

Example Code:

```
class FileManager:
    def __init__(self, filename, mode):
        self.file = open(filename, mode)

    def __enter__(self):
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
```

```
        self.file.close()

with FileManager("test.txt", "w") as f:
    f.write("Hello, world!")

# File is automatically closed after `with` block
```

---

10. What is __del__ method in Python?

Answer:

__del__ is a destructor, called when an object is deleted or goes out of scope.

Example Code:

```
class Demo:
    def __init__(self):
        print("Object Created")

    def __del__(self):
        print("Object Destroyed")

obj = Demo()
del obj  # Output: Object Destroyed
```

Key Takeaways

- ☑ Dunder methods allow customization of object behavior.
- ☑ __init__, __str__, __repr__, __call__, and __len__ are commonly used.
- ☑ __add__, __eq__, and __getitem__ enable operator overloading.
- ☑ __enter__ and __exit__ help in context management (with statement).
- ☑ __del__ acts as a destructor.

Metaclasses

---

1. What is a metaclass in Python?

Answer:

A metaclass is a class that defines the behavior of other classes. It allows us to modify class creation dynamically.

Example Code:

```
class Meta(type):
    def __new__(cls, name, bases, dct):
        print(f"Creating class: {name}")
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=Meta):
    pass

# Output: Creating class: MyClass
```

---

2. How does a metaclass differ from a regular class?

Answer:

- A regular class creates objects (instances).
- A metaclass creates classes.

Example Code:

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

obj = MyClass()  # Regular object
print(type(obj))  # Output: <class '__main__.MyClass'>
```

```
print(type(MyClass))  # Output: <class '__main__.Meta'> (metaclass)
```

📌 MyClass is an instance of Meta, and obj is an instance of MyClass.

---

3. How do you define a custom metaclass using __new__?

Answer:

The __new__ method modifies class creation before instantiation.

Example Code:

```
class CustomMeta(type):
    def __new__(cls, name, bases, dct):
        dct['custom_attr'] = "Added by metaclass"
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=CustomMeta):
    pass

print(MyClass.custom_attr)  # Output: Added by metaclass
```

📌 The metaclass adds custom_attr dynamically to MyClass.

---

4. What is the purpose of __init__ in a metaclass?

Answer:

- __new__ creates the class.
- __init__ initializes the class after creation.

Example Code:

```
class Meta(type):
    def __init__(cls, name, bases, dct):
        print(f"Initializing class: {name}")
        super().__init__(name, bases, dct)

class MyClass(metaclass=Meta):
```

```
    pass


# Output:
# Initializing class: MyClass
```

---

5. How can a metaclass enforce coding rules?

Answer:

We can prevent method names from being lowercase, enforce attributes, or raise errors if a rule is violated.

Example Code:

```
class EnforceMeta(type):
    def __new__(cls, name, bases, dct):
        for attr in dct:
            if attr.islower():
                raise TypeError(f"Attribute {attr} must be uppercase")
        return super().__new__(cls, name, bases, dct)


class MyClass(metaclass=EnforceMeta):
    CONSTANT = 42
    def METHOD(self): pass  # Valid


# class InvalidClass(metaclass=EnforceMeta):
#     invalid_attr = 10  # Raises TypeError
```

📌 The metaclass prevents lowercase attributes, ensuring consistency.

---

6. Can you use metaclasses to add methods to a class?

Answer:

Yes, metaclasses can dynamically add methods.

Example Code:

```
class AddMethodMeta(type):
    def __new__(cls, name, bases, dct):
```

```
        def new_method(self):
            return "New method added by metaclass!"
        dct['new_method'] = new_method
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=AddMethodMeta):
    pass

obj = MyClass()
print(obj.new_method())  # Output: New method added by metaclass!
```

📌 The method new_method was added dynamically by the metaclass.

---

7. How does __call__ work in metaclasses?

Answer:

__call__ in a metaclass controls object instantiation.

Example Code:

```
class Meta(type):
    def __call__(cls, *args, **kwargs):
        print(f"Creating an instance of {cls.__name__}")
        return super().__call__(*args, **kwargs)

class MyClass(metaclass=Meta):
    pass

obj = MyClass()
# Output: Creating an instance of MyClass
```

📌 __call__ is triggered before an object is created.

---

8. How do metaclasses help in singleton design patterns?

Answer:

Metaclasses can restrict class instantiation to a single instance.

Example Code:

```
class SingletonMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class Singleton(metaclass=SingletonMeta):
    pass

obj1 = Singleton()
obj2 = Singleton()
print(obj1 is obj2)  # Output: True (Both refer to the same instance)
```

📌 The singleton pattern ensures only one instance exists.

---

9. Can metaclasses be inherited?

Answer:

Yes, metaclasses can be inherited, affecting subclass creation.

Example Code:

```
class BaseMeta(type):
    def __new__(cls, name, bases, dct):
        dct['base_attr'] = "Inherited from metaclass"
        return super().__new__(cls, name, bases, dct)

class Parent(metaclass=BaseMeta):
    pass
```

```
class Child(Parent):
    pass

print(Child.base_attr)  # Output: Inherited from metaclass
```

📌 Child inherits the metaclass behavior from Parent.

---

10. How do you check the metaclass of a class?

Answer:

Use type(ClassName) to check the metaclass.

Example Code:

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

print(type(MyClass))  # Output: <class '__main__.Meta'>
```

📌 MyClass is an instance of Meta.

Key Takeaways

- ☑ Metaclasses control how classes are created.
- ☑ __new__ modifies class creation before instantiation.
- ☑ __call__ modifies instance creation dynamically.
- ☑ Singleton, method injection, and coding rules can be enforced.
- ☑ Metaclasses can be inherited, allowing custom class behaviors.

Decorators

---

1. What is a decorator in Python?

Answer:

A decorator is a function that modifies the behavior of another function or method without changing its code. It is often used for logging, authentication, caching, and timing execution.

Example Code:

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

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

say_hello()
```

Output:

```
Before function execution
Hello, World!
After function execution
```

📌 The wrapper function modifies say_hello() without altering its code.

---

2. How do function decorators work in Python?

Answer:

A function decorator wraps another function using @decorator_name and extends its behavior.

Example Code:

```
def uppercase_decorator(func):
    def wrapper():
        result = func()
```

```
        return result.upper()
    return wrapper

@uppercase_decorator
def greet():
    return "hello"

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

📌 The uppercase_decorator modifies greet() by converting its output to uppercase.

---

3. How do you pass arguments to a decorator?

Answer:

To pass arguments, use *args and **kwargs inside the wrapper function.

Example Code:

```
def repeat_decorator(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat_decorator(times=3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

Output:

```
Hello, Alice!
Hello, Alice!
```

Hello, Alice!

📌 The repeat_decorator takes arguments and modifies the function accordingly.

---

4. How do you decorate functions with return values?

Answer:

Use return inside the wrapper function to return the modified output.

Example Code:

```
def add_prefix(func):
    def wrapper(name):
        return "Mr. " + func(name)
    return wrapper

@add_prefix
def get_name(name):
    return name

print(get_name("John"))  # Output: Mr. John
```

📌 The decorator modifies the return value without altering the original function.

---

5. How do you apply multiple decorators to a function?

Answer:

Multiple decorators are applied from bottom to top.

Example Code:

```
def bold_decorator(func):
    def wrapper():
        return "<b>" + func() + "</b>"
    return wrapper

def italic_decorator(func):
    def wrapper():
```

```
        return "<i>" + func() + "</i>"
    return wrapper

@bold_decorator
@italic_decorator
def text():
    return "Hello"

print(text())  # Output: <b><i>Hello</i></b>
```

📌 The italic_decorator applies first, followed by the bold_decorator.

---

6. What are class decorators, and how do they work?

Answer:

A class decorator is a class that modifies functions using the __call__ method.

Example Code:

```
class DecoratorClass:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print("Before function execution")
        result = self.func(*args, **kwargs)
        print("After function execution")
        return result

@DecoratorClass
def hello():
    print("Hello, World!")

hello()
```

Output:

Before function execution

Hello, World!

After function execution

📌 The __call__ method makes the class act like a function decorator.

---

7. How do decorators work with methods in a class?

Answer:

Use decorators for logging, validation, or access control in class methods.

Example Code:

```
def method_decorator(func):
    def wrapper(self, *args, **kwargs):
        print(f"Calling method {func.__name__}")
        return func(self, *args, **kwargs)
    return wrapper

class MyClass:
    @method_decorator
    def greet(self):
        print("Hello from MyClass")

obj = MyClass()
obj.greet()
```

Output:

Calling method greet

Hello from MyClass

📌 The decorator logs method calls without modifying greet().

---

8. How do you create a decorator to measure function execution time?

Answer:

Use the time module to measure execution time.

Example Code:

```
import time

def timing_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} executed in {end_time - start_time:.6f} seconds")
        return result
    return wrapper

@timing_decorator
def slow_function():
    time.sleep(2)
    print("Function finished")

slow_function()
```

Output:

```
Function finished
slow_function executed in 2.000xxx seconds
```

📌 This decorator measures the execution time of a function.

---

9. How do you create a decorator for access control?

Answer:

A decorator can restrict access based on conditions like user roles.

Example Code:

```
def admin_only(func):
    def wrapper(user):
```

```
        if user != "admin":
            print("Access denied!")
        else:
            return func(user)
    return wrapper

@admin_only
def view_dashboard(user):
    print(f"Welcome {user}, you have access to the dashboard.")

view_dashboard("guest")  # Output: Access denied!
view_dashboard("admin")  # Output: Welcome admin, you have access to the dashboard.
```

📌 The admin_only restricts access based on the user role.

---

10. What is functools.wraps and why is it used in decorators?

Answer:

functools.wraps preserves the original function metadata when using decorators.

Example Code:

```
from functools import wraps

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

@log_decorator
def my_function():
    """This is my function"""
```

```
    print("Hello!")


print(my_function.__name__)  # Output: my_function
print(my_function.__doc__)  # Output: This is my function
```

📌 Without @wraps, my_function.__name__ would return "wrapper" instead of "my_function".

Key Takeaways

☑ Decorators modify function behavior without changing the source code.
☑ Use @decorator_name for function decorators.
☑ Multiple decorators can be stacked.
☑ Class decorators use __call__.
☑ functools.wraps preserves function metadata.
☑ Common use cases include logging, access control, execution timing, and method validation.

Generators

---

1. What is a generator in Python?

Answer:

A generator is a special type of iterator that allows you to iterate over data without storing it in memory. It is created using a function with the yield statement.

Example Code:

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

```
gen = simple_generator()
print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

📌 Unlike lists, generators generate values on the fly and do not store them in memory.

---

2. What is the difference between yield and return in Python?

Answer:

- return terminates a function and returns a value.
- yield pauses the function and returns a value but allows resuming execution.

Example Code:

```
def yield_example():
    yield "Hello"
    yield "World"

gen = yield_example()
print(next(gen))  # Output: Hello
print(next(gen))  # Output: World
```

📌 The function remembers its state after yield, allowing execution to continue from the last pause.

---

3. How do you create an infinite generator?

Answer:

An infinite generator can be created using an infinite loop with yield.

Example Code:

```
def infinite_numbers():
    num = 1
    while True:
        yield num
        num += 1
```

```
gen = infinite_numbers()
print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

📌 The generator keeps yielding values indefinitely without memory overhead.

---

4. What are the advantages of using generators?

Answer:

1. Memory Efficient: Generates values on-demand.
2. Lazy Evaluation: Computes values only when needed.
3. Improves Performance: Avoids storing large datasets in memory.
4. Simpler Code: No need to manage iteration manually.

Example Code:

```
import sys

# Using a list
numbers_list = [i for i in range(1000000)]
print(sys.getsizeof(numbers_list), "bytes")  # Large memory usage

# Using a generator
def numbers_generator():
    for i in range(1000000):
        yield i

gen = numbers_generator()
print(sys.getsizeof(gen), "bytes")  # Very small memory usage
```

📌 Generators consume less memory compared to lists.

---

5. Can a generator be restarted?

Answer:

No, a generator cannot be restarted. Once exhausted, a new instance must be created.

Example Code:

```
def my_generator():
    yield 1
    yield 2


gen = my_generator()
print(list(gen))  # Output: [1, 2]
print(list(gen))  # Output: [] (generator exhausted)

# Creating a new instance
gen = my_generator()
print(list(gen))  # Output: [1, 2]
```

📌 A generator does not reset automatically, a new instance must be created.

---

6. How do you use send() in generators?

Answer:

The send() method sends a value into a generator and resumes execution.

Example Code:

```
def my_generator():
    value = yield "Start"
    yield f"Received: {value}"


gen = my_generator()
print(next(gen))     # Output: Start
print(gen.send(100))  # Output: Received: 100
```

📌 send(value) resumes execution while passing a value to yield.

---

7. How do you handle exceptions inside a generator?

Answer:

Use try-except inside the generator to handle exceptions gracefully.

Example Code:

```
def error_handling_generator():
    try:
        yield 1
        yield 2
        yield 3
    except GeneratorExit:
        print("Generator closed")


gen = error_handling_generator()
print(next(gen))  # Output: 1
gen.close()       # Closes the generator
```

📌 The GeneratorExit exception is raised when close() is called on a generator.

---

8. How do you convert a generator to a list?

Answer:

Use the list() function to convert a generator into a list.

Example Code:

```
def number_generator():
    for i in range(5):
        yield i


gen = number_generator()
print(list(gen))  # Output: [0, 1, 2, 3, 4]
```

📌 Converting a generator to a list stores all values in memory, losing the benefits of lazy evaluation.

---

9. What is a generator expression, and how is it different from a list comprehension?

Answer:

A generator expression is like a list comprehension but uses parentheses () instead of square brackets [].

Example Code:

```
# List comprehension (stores all values in memory)
list_comp = [i * 2 for i in range(5)]
print(list_comp)  # Output: [0, 2, 4, 6, 8]

# Generator expression (efficient)
gen_exp = (i * 2 for i in range(5))
print(next(gen_exp))  # Output: 0
print(next(gen_exp))  # Output: 2
```

📌 Generators are memory-efficient compared to list comprehensions.

---

10. What is the difference between an iterator and a generator?

Answer:

Feature	Iterator	Generator
Creation	Uses __iter__() and __next__()	Uses yield
Memory Usage	Stores all values	Generates values on-demand
Resumability	No	Yes
Complexity	Requires manual implementation	Simplifies iteration

Example Code (Iterator vs Generator):

```
# Iterator
class MyIterator:
    def __init__(self, max):
        self.max = max
        self.current = 0

    def __iter__(self):
        return self
```

```
    def __next__(self):
        if self.current < self.max:
            self.current += 1
            return self.current
        else:
            raise StopIteration

it = MyIterator(3)
print(list(it))  # Output: [1, 2, 3]

# Generator (simpler way)
def my_generator(max):
    for i in range(1, max + 1):
        yield i

gen = my_generator(3)
print(list(gen))  # Output: [1, 2, 3]
```

📌 Generators are easier to implement than iterators.

Key Takeaways

☑ Generators provide a memory-efficient way to iterate over data.
☑ yield pauses execution and remembers the function's state.
☑ send() allows sending values into a generator.
☑ Generators cannot be restarted, you need to create a new instance.
☑ Generator expressions ((expr for item in iterable)) are more memory-efficient than list comprehensions.
☑ Use generators for large datasets, streaming data, and lazy evaluation.

Context Managers

---

1. What is a context manager in Python?

Answer:

A context manager is an object that defines the methods __enter__() and __exit__() to manage resources properly.

Example Code:

```
class MyContextManager:
    def __enter__(self):
        print("Entering the context")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context")

with MyContextManager():
    print("Inside the context block")
```

Output:

```
Entering the context
Inside the context block
Exiting the context
```

📌 Ensures proper cleanup after the with block execution.

---

2. How does a context manager handle exceptions?

Answer:

If an exception occurs inside the with block, the __exit__() method handles it.

Example Code:

```
class ExceptionHandlingCM:
    def __enter__(self):
        print("Entering context")

    def __exit__(self, exc_type, exc_value, traceback):
```

```
        print("Exception handled:", exc_type)
        return True  # Suppresses exception

with ExceptionHandlingCM():
    print(1 / 0)  # Division by zero
print("Code continues...")
```

Output:

```
Entering context
Exception handled: <class 'ZeroDivisionError'>
Code continues...
```

📌 If __exit__() returns True, the exception is suppressed.

---

3. How do you use the with statement with files?

Answer:

The with statement ensures automatic file closing.

Example Code:

```
with open("example.txt", "w") as file:
    file.write("Hello, world!")
```

📌 No need to explicitly call file.close().

---

4. What happens if an exception occurs inside a with block?

Answer:

The __exit__() method is always called, even if an exception occurs.

Example Code:

```
class MyContext:
    def __enter__(self):
        print("Start")

    def __exit__(self, exc_type, exc_value, traceback):
        print("Cleanup even if an error occurs")
```

```
with MyContext():
    raise ValueError("Something went wrong")
```

📌 The cleanup still happens, even with an exception.

---

5. How do you create a context manager using contextlib?

Answer:

The contextlib module allows simpler context manager creation using @contextmanager.

Example Code:

```
from contextlib import contextmanager

@contextmanager
def my_context():
    print("Entering context")
    yield
    print("Exiting context")

with my_context():
    print("Inside block")
```

📌 Easier than defining a class with __enter__() and __exit__().

---

6. Can a context manager return a value?

Answer:

Yes! The __enter__() method can return a resource.

Example Code:

```
class FileManager:
    def __enter__(self):
        self.file = open("test.txt", "w")
        return self.file
```

```
    def __exit__(self, exc_type, exc_value, traceback):
        self.file.close()

with FileManager() as file:
    file.write("Hello, Python!")
```

📌 The __enter__() method returns the file object.

---

7. What is the difference between a class-based and a function-based context manager?

Answer:

Feature	Class-Based	Function-Based (contextlib)
Implementation	Uses __enter__() and __exit__()	Uses @contextmanager
Verbosity	More code	Less code
Flexibility	More control	Simpler

Example Code:

Class-Based:

```
class CM:
    def __enter__(self):
        print("Start")

    def __exit__(self, exc_type, exc_value, traceback):
        print("End")
```

Function-Based:

```
from contextlib import contextmanager

@contextmanager
def cm():
    print("Start")
    yield
    print("End")
```

📌 Use contextlib for simpler cases.

---

8. Can you nest context managers?

Answer:

Yes, multiple context managers can be nested.

Example Code:

```
with open("file1.txt", "w") as f1, open("file2.txt", "w") as f2:
    f1.write("Hello")
    f2.write("World")
```

📌 Multiple files are handled in a single with statement.

---

9. How do you create a context manager for managing database connections?

Answer:

A database connection must always be closed properly.

Example Code:

```
import sqlite3
from contextlib import contextmanager

@contextmanager
def db_connection(db_name):
    conn = sqlite3.connect(db_name)
    cursor = conn.cursor()
    try:
        yield cursor
    finally:
        conn.commit()
        conn.close()

with db_connection("test.db") as cursor:
    cursor.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER, name TEXT)")
```

📌 Ensures the connection is closed even in case of failure.

10. How do you use a context manager to temporarily change the working directory?

Answer:

The chdir() method in os can be wrapped in a context manager.

Example Code:

```
import os
from contextlib import contextmanager

@contextmanager
def change_directory(path):
    original_path = os.getcwd()
    os.chdir(path)
    try:
        yield
    finally:
        os.chdir(original_path)

with change_directory("/tmp"):
    print("Current Directory:", os.getcwd())
```

📌 Automatically reverts to the original directory after exiting the block.

Key Takeaways

- ☑ Context managers ensure proper resource management.
- ☑ The with statement automatically handles cleanup.
- ☑ contextlib provides a simpler way to create context managers.
- ☑ __exit__() handles exceptions and ensures cleanup.
- ☑ Nested context managers help manage multiple resources.
