PYTHON

INTERVIEW QUESTIONS

A collection of Python questions for technical interview preparation

40

questions

TABLE OF CONTENTS

Question 1

What is Python and what are its key features?

> ***Python*** *is a high-level, interpreted, general-purpose programming language with a focus on code readability and simple syntax.*

Key features:

- **Interpreted**: code is executed line by line by the interpreter, without the need for compilation.
- **Dynamic typing**: variable types are determined automatically at runtime, not at declaration.
- **Automatic memory management**: a built-in Garbage Collector frees unused memory.
- **Cross-platform**: code runs on Windows, macOS, and Linux without modifications.
- **Multi-paradigm**: supports object-oriented, functional, and procedural programming styles.
- **Extensive standard library**: modules for working with files, networking, JSON, and much more.

Where it's used:

- **Web development**: Django, Flask, FastAPI.
- **Data analysis and ML**: NumPy, Pandas, scikit-learn.
- **Automation and scripting**: file processing, DevOps, testing.

Question 2

What are the basic data types in Python?

Python provides several built-in data types:

Numeric types:

- `int` — integers of arbitrary precision: `42`, `-7`, `1_000_000`
- `float` — floating-point numbers: `3.14`, `-0.5`, `1e10`
- `complex` — complex numbers: `3+4j`

Text type:

- `str` — character string: `"Hello"`, `'Python'`

Boolean type:

- `bool` — takes values `True` or `False`

Special type:

- `NoneType` — the single value `None`, meaning the absence of a value

Type conversion:

```
# Explicit type conversion
x = int("42")       # str → int
y = float(42)       # int → float
z = str(3.14)       # float → str
w = bool(0)          # int → bool (False)

# Type checking
print(type(x))  # <class 'int'>
```

Question 3

What is the difference between a list and a tuple?

List:

- **Mutable**: you can add, remove, and modify elements.
- Created using square brackets `[]`.
- Uses more memory due to resizing capability.

```
fruits = ["apple", "banana", "cherry"]
fruits.append("pear")       # Add element
fruits[0] = "orange"        # Modify element
```

Tuple:

- **Immutable**: cannot be changed after creation.
- Created using parentheses `()`.
- Faster and uses less memory.
- Can be used as a dictionary key (since it's hashable).

```
point = (10, 20)
# point[0] = 5  # TypeError - cannot modify

# Tuple as a dictionary key
locations = {(55.75, 37.62): "Moscow"}
```

When to use which:

- `list` — when the collection will change (adding, removing elements).
- `tuple` — when data should be immutable (coordinates, configurations, dictionary keys).

Question 4

What is the difference between set and frozenset?

set:

- A **mutable** unordered collection of unique elements.
- Supports adding and removing elements.
- Cannot be used as a dictionary key or element of another set.

```
colors = {"red", "green", "blue"}
colors.add("yellow")
colors.discard("red")
```

frozenset:

- An **immutable** version of a set.
- Does not support adding or removing elements.
- Is hashable — can be used as a dictionary key.

```
immutable_set = frozenset([1, 2, 3])
# immutable_set.add(4)  # AttributeError

# frozenset as a dictionary key
cache = {frozenset([1, 2]): "result"}
```

Common set operations:

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}

a | b   # Union: {1, 2, 3, 4, 5, 6}
a & b   # Intersection: {3, 4}
a - b   # Difference: {1, 2}
a ^ b   # Symmetric difference: {1, 2, 5, 6}
```

In practice:

`set` is used for fast deduplication and membership checks (`in` runs in O(1)). `frozenset` is needed when a set must serve as a dictionary key or an element of another set.

Question 5

What is a dictionary (dict) and how does it work?

> ***Dictionary (dict)*** *is a mutable collection of key-value pairs implemented using a hash table.*

Key characteristics:

- Access, insertion, and deletion are performed in **O(1)** on average.
- Keys must be **hashable** (strings, numbers, tuples).
- Since Python 3.7, dictionaries **preserve insertion order**.

```
user = {
    "name": "Anna",
    "age": 25,
    "city": "Moscow"
}
```

Main methods:

```
user["name"]              # Access by key (KeyError if missing)
user.get("email", "-")    # Safe access with default value

user.keys()               # All keys
user.values()             # All values
user.items()              # Key-value pairs

user.pop("city")          # Remove and return value
user.update({"age": 26})  # Update values
```

Creating a dictionary:

```
# Literal
d1 = {"a": 1, "b": 2}

# From a list of tuples
d2 = dict([("a", 1), ("b", 2)])

# Using dict comprehension
d3 = {x: x ** 2 for x in range(5)}
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Question 6

What are mutable and immutable data types?

> *In Python, all objects are divided into **mutable** and **immutable** depending on whether their contents can be changed after creation.*

Immutable:

- `int`, `float`, `bool`
- `str`
- `tuple`
- `frozenset`

When "modified," a new object is created:

```
x = 10
print(id(x))  # e.g.: 140234866357520
x += 1
print(id(x))  # Different id - this is a new object
```

Mutable:

- `list`
- `dict`
- `set`

The object is modified "in place":

```
lst = [1, 2, 3]
print(id(lst))  # e.g.: 140234866400064
lst.append(4)
print(id(lst))  # Same id - object was modified
```

Why this matters:

- **Dictionary keys** can only be immutable objects.
- **Passing to functions**: mutable objects can be changed inside a function, which may cause unexpected side effects.
- **Default values**: don't use mutable objects as default values in functions.

```
# Common mistake
def add_item(item, lst=[]):  # Same list across all calls!
    lst.append(item)
    return lst

# Correct approach
def add_item(item, lst=None):
    if lst is None:
        lst = []
    lst.append(item)
    return lst
```

Question 7

How do indexing and slicing work in Python?

Indexing:

- Elements are numbered starting from `0`.
- Negative indices count from the end: `-1` is the last element.

```
text = "Python"
text[0]    # 'P'
text[-1]   # 'n'
text[-2]   # 'o'
```

Slicing:

Syntax: `[start:stop:step]`

- `start` — starting index (inclusive), defaults to `0`.
- `stop` — ending index (exclusive), defaults to the length of the sequence.
- `step` — step size, defaults to `1`.

```
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

nums[2:5]     # [2, 3, 4]
nums[:3]      # [0, 1, 2]
nums[7:]      # [7, 8, 9]
nums[::2]     # [0, 2, 4, 6, 8]  - every second
nums[::-1]    # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]  - reverse
```

Slicing works with strings and tuples:

```
text = "Hello, World!"
text[7:12]    # 'World'
text[::-1]    # '!dlroW ,olleH'

coords = (10, 20, 30, 40, 50)
coords[1:4]   # (20, 30, 40)
```

Question 8

What are f-strings and how do they compare to other formatting methods?

Python offers several ways to format strings:

f-strings — recommended approach:

Available since Python 3.6. Allow embedding expressions directly in the string.

```
name = "Anna"
age = 25
print(f"Hello, {name}! You are {age} years old.")
print(f"In 5 years you will be {age + 5}.")
print(f"Name in uppercase: {name.upper()}")
```

.format() method:

```
print("Hello, {}! You are {} years old.".format(name, age))
print("Hello, {0}! {0}, you are {1} years old.".format(name, age))
```

% operator (legacy):

```
print("Hello, %s! You are %d years old." % (name, age))
```

Comparison:

- **f-strings** — the most readable and fastest method. Supports any expressions.
- **.format()** — useful when the string template is defined in advance.
- **%** — legacy approach, found in older code.

Number formatting:

```
pi = 3.14159265
print(f"Pi: {pi:.2f}")          # Pi: 3.14
print(f"Number: {1000000:,}")   # Number: 1,000,000
print(f"Percent: {0.856:.1%}")  # Percent: 85.6%
```

Question 9

How do if / elif / else conditional statements work?

Conditional statements allow you to execute different blocks of code depending on a condition.

Syntax:

```
age = 18

if age < 13:
    print("Child")
elif age < 18:
    print("Teenager")
else:
    print("Adult")
```

Ternary operator:

A concise way to write a condition in a single line:

```
status = "adult" if age >= 18 else "minor"
```

Truthy and Falsy values:

In Python, the following values are considered **false** (Falsy):

- `False`, `None`
- `0`, `0.0`
- Empty collections: `""`, `[]`, `()`, `{}`, `set()`

Everything else is considered **true** (Truthy):

```
items = []

if items:
    print("List is not empty")
else:
    print("List is empty")  # This block will execute
```

Chained comparisons:

Python supports chained comparisons:

```
x = 5
if 1 < x < 10:
    print("x is in the range from 1 to 10")
```

Question 10

What kind of loops exist in Python and how do they differ?

for loop:

Iterates over the elements of an **iterable object** (list, string, range, etc.):

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

# Loop over a range of numbers
for i in range(5):
    print(i)  # 0, 1, 2, 3, 4
```

while loop:

Executes **as long as the condition is true**:

```
count = 0
while count < 3:
    print(count)
    count += 1
```

Control statements:

- `break` — terminates the loop.
- `continue` — skips to the next iteration.

```
for i in range(10):
    if i == 3:
        continue  # Skips 3
    if i == 7:
        break     # Stops the loop at 7
    print(i)      # 0, 1, 2, 4, 5, 6
```

else block in a loop:

Executes if the loop completes **without** encountering a `break`:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            break
    else:
        # Executes if break was not triggered
        print(f"{n} is a prime number")
```

Question 11

What is a function in Python and how do you define it?

> *A **function** is a named block of code that can be called multiple times. Functions help avoid duplication and make the code more readable.*

Definition and calling:

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

message = greet("Anna")
print(message)  # Hello, Anna!
```

Parameters and arguments:

```
# Default values
def power(base, exponent=2):
    return base ** exponent

power(3)     # 9  (exponent = 2)
power(3, 3)  # 27 (exponent = 3)
```

Keyword arguments:

```
def create_user(name, age, city="Moscow"):
    return {"name": name, "age": age, "city": city}

# Keyword arguments can be passed in any order
user = create_user(age=25, name="Anna")
```

Returning multiple values:

```
def min_max(numbers):
    return min(numbers), max(numbers)

lo, hi = min_max([3, 1, 7, 2, 9])
print(lo, hi)  # 1 9
```

Function without return:

If `return` is absent, the function returns `None`:

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

result = say_hello("World")
print(result)  # None
```

Question 12

What are *args and **kwargs?

args — arbitrary number of positional arguments:

*Collects all extra positional arguments into a **tuple**:*

```
def total(*args):
    return sum(args)

total(1, 2, 3)      # 6
total(10, 20)        # 30
```

kwargs — arbitrary number of keyword arguments:

***Collects all extra keyword arguments into a** dictionary:*

```
def build_profile(**kwargs):
    return kwargs

build_profile(name="Anna", age=25, city="Moscow")
# {'name': 'Anna', 'age': 25, 'city': 'Moscow'}
```

Combining:

The order of parameters in a function definition is strictly fixed: regular → `args` → keyword-only → `**kwargs` :

```
def func(a, b, *args, **kwargs):
    print(f"a={a}, b={b}")
    print(f"args={args}")
    print(f"kwargs={kwargs}")

func(1, 2, 3, 4, x=10, y=20)
# a=1, b=2
# args=(3, 4)
# kwargs={'x': 10, 'y': 20}
```

Unpacking during a call:

```
def greet(name, age):
    print(f"{name}, {age} years old")

args_list = ["Anna", 25]
greet(*args_list)      # Unpacking a list

kwargs_dict = {"name": "Ivan", "age": 30}
greet(**kwargs_dict)   # Unpacking a dictionary
```

Question 13

What is an anonymous function (lambda)?

> ***Lambda*** *is an anonymous (unnamed) function defined in a single line. It can take any number of arguments but contains only one expression.*

Syntax:

```
# Regular function
def square(x):
    return x ** 2

# Equivalent lambda
square = lambda x: x ** 2

square(5)  # 25
```

Usage with sorted:

```
users = [
    {"name": "Anna", "age": 25},
    {"name": "Boris", "age": 30},
    {"name": "Vera", "age": 20},
]

# Sort by age
sorted_users = sorted(users, key=lambda u: u["age"])
```

Usage with map and filter:

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

squares = list(map(lambda x: x ** 2, numbers))
# [1, 4, 9, 16, 25]

evens = list(filter(lambda x: x % 2 == 0, numbers))
# [2, 4]
```

Limitations:

- Only **one expression** — you cannot use multi-line logic, loops, or assignments.

- Reduces **readability** for complex expressions — it's better to use a regular function.
- No **name** — makes debugging harder (appears as `<lambda>` in the traceback).

Question 14

What are variable scopes in Python?

> *In Python, scope determines where a variable is accessible. Python uses the **LEGB** rule for resolving variable names.*

The LEGB Rule:

- **L — Local**: variables inside the current function.
- **E — Enclosing**: variables in the outer (enclosing) function.
- **G — Global**: variables at the module level.
- **B — Built-in**: Python's built-in names (`print` , `len` , `range`).

```
x = "global"  # Global

def outer():
    x = "enclosing"  # Enclosing

    def inner():
        x = "local"  # Local
        print(x)  # "local"

    inner()

outer()
```

The global keyword:

Allows modifying a global variable inside a function:

```
counter = 0

def increment():
    global counter
    counter += 1

increment()
print(counter)  # 1
```

The nonlocal keyword:

Allows modifying a variable from an outer (enclosing) function:

```
def outer():
    count = 0

    def inner():
        nonlocal count
        count += 1
        return count

    return inner

counter = outer()
print(counter())  # 1
print(counter())  # 2
```

Question 15

What is list comprehension and how to use it?

> ***List comprehension*** *is a concise way to create a new list from an existing collection in a single line.*

Syntax:

```
[expression for item in iterable]
```

Examples:

```
# Squares of numbers
squares = [x ** 2 for x in range(6)]
# [0, 1, 4, 9, 16, 25]

# Equivalent with a loop
squares = []
for x in range(6):
    squares.append(x ** 2)
```

With condition (filtering):

```
# Only even numbers
evens = [x for x in range(10) if x % 2 == 0]
# [0, 2, 4, 6, 8]
```

With if/else condition (transformation):

```
labels = ["even" if x % 2 == 0 else "odd" for x in range(5)]
# ['even', 'odd', 'even', 'odd', 'even']
```

Nested comprehensions:

```
# Multiplication table
matrix = [[i * j for j in range(1, 4)] for i in range(1, 4)]
# [[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```

Analogs for other types:

```
# Dict comprehension
squares_dict = {x: x ** 2 for x in range(5)}

# Set comprehension
unique_lengths = {len(word) for word in ["cat", "dog", "fox"]}
```

Important:

Avoid overusing complex nested comprehensions — if the expression is hard to read, use a regular loop instead.

Question 16

What are generators and how do they differ from lists?

> *A **generator** is a function that returns items one by one using `yield`, rather than creating the entire list in memory all at once.*

Generator function:

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1

for num in count_up_to(5):
    print(num)  # 1, 2, 3, 4, 5
```

Generator expression:

Similar to a list comprehension, but uses parentheses:

```
# List comprehension - creates the whole list in memory
squares_list = [x ** 2 for x in range(1000000)]

# Generator expression - evaluates one by one
squares_gen = (x ** 2 for x in range(1000000))
```

Key differences from a list:

- **Memory**: a generator only stores the current item, not the entire collection.
- **Single use**: a generator can be iterated over only once.
- **Laziness**: items are evaluated on demand, not in advance.

```
gen = (x for x in range(3))
print(list(gen))  # [0, 1, 2]
print(list(gen))  # [] - already exhausted
```

When to use:

- **Generator** — when working with large datasets, streams, or when there is no need to keep all items.
- **List** — when multiple access, indexing is required, or the dataset is known to be small.

Question 17

How do the built-in map, filter, and zip functions work?

map — applies a function to each item:

```
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x ** 2, numbers))
# [1, 4, 9, 16, 25]

# Equivalent using comprehension
squares = [x ** 2 for x in numbers]
```

filter — selects items by condition:

```
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
# [2, 4, 6]

# Equivalent using comprehension
evens = [x for x in numbers if x % 2 == 0]
```

zip — combines multiple collections:

```
names = ["Anna", "Boris", "Vera"]
ages = [25, 30, 22]

pairs = list(zip(names, ages))
# [('Anna', 25), ('Boris', 30), ('Vera', 22)]

# Often used to create a dictionary
user_ages = dict(zip(names, ages))
# {'Anna': 25, 'Boris': 30, 'Vera': 22}
```

Features:

- All three functions return **iterators**, not lists — you need to wrap them in `list()` to get a list.
- `zip` stops according to the **shortest** collection.
- In most cases, **list comprehension** is considered a more readable alternative to `map` and `filter`.

Question 18

What is unpacking in Python?

> ***Unpacking*** *is a mechanism that allows you to "extract" a collection into separate variables.*

Multiple assignment:

```
a, b, c = [1, 2, 3]
print(a, b, c)  # 1 2 3

# Works with tuples, strings, and other iterables
x, y = (10, 20)
first, second, third = "abc"
```

Variable swapping (swap):

```
a, b = 1, 2
a, b = b, a
print(a, b)  # 2 1
```

Unpacking with * (asterisk):

Collects the "remaining" items into a list:

```
first, *rest = [1, 2, 3, 4, 5]
print(first)  # 1
print(rest)   # [2, 3, 4, 5]

first, *middle, last = [1, 2, 3, 4, 5]
print(middle)  # [2, 3, 4]
```

Unpacking in function calls:

```
def greet(name, age, city):
    print(f"{name}, {age}, {city}")

data = ["Anna", 25, "Moscow"]
greet(*data)  # Unpacking a list

info = {"name": "Ivan", "age": 30, "city": "St. Petersburg"}
greet(**info)  # Unpacking a dictionary
```

Unpacking in nested structures:

```
points = [(1, 2), (3, 4), (5, 6)]
for x, y in points:
    print(f"x={x}, y={y}")
```

Question 19

How does exception handling work in Python?

> *An **exception** is an error that occurs during the execution of a program. Python allows catching and handling exceptions so the program doesn't crash.*

try/except syntax:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Division by zero!")
```

Full try/except/else/finally construct:

```
try:
    number = int(input("Enter a number: "))
except ValueError:
    print("That's not a number!")
else:
    # Executes if NO exception occurred
    print(f"You entered: {number}")
finally:
    # ALWAYS executes
    print("Shutting down")
```

Catching multiple exceptions:

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

Hierarchy of main exceptions:

- `BaseException`

- `Exception` — base class for most exceptions - `ValueError` — invalid value - `TypeError` — invalid type - `KeyError` — key not found in dictionary - `IndexError` — index out of range for a list - `FileNotFoundError` — file not found - `ZeroDivisionError` — division by zero

Important rule:

Catch **specific** exceptions instead of a generic `except Exception` — this helps avoid hiding unexpected errors.

Question 20

How to create a custom exception?

> *Custom exceptions are created by inheriting from the `Exception` class (or its subclasses).*

Simple custom exception:

```
class InsufficientFundsError(Exception):
    pass

def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientFundsError("Insufficient funds")
    return balance - amount

try:
    withdraw(100, 200)
except InsufficientFundsError as e:
    print(e)  # Insufficient funds
```

Exception with additional data:

```
class ValidationError(Exception):
    def __init__(self, field, message):
        self.field = field
        self.message = message
        super().__init__(f"{field}: {message}")

try:
    raise ValidationError("email", "Invalid format")
except ValidationError as e:
    print(e.field)    # email
    print(e.message)  # Invalid format
```

When to create custom exceptions:

- When built-in exceptions don't describe the error accurately enough.
- To separate business logic errors (e.g., `UserNotFoundError`, `PermissionDeniedError`).
- To easily catch a group of related errors using a common base class.

```
class AppError(Exception):
    """Base application exception"""
    pass

class NotFoundError(AppError):
    pass

class AccessDeniedError(AppError):
    pass
```

Question 21

What is a context manager and how does the with statement work?

> *A **context manager** is an object that automatically performs actions when entering a block of code and when exiting it (even if an error occurs).*

The with statement:

The most common example is working with files:

```
# Without with - you must remember to close the file
file = open("data.txt", "r")
try:
    content = file.read()
finally:
    file.close()

# With with - the file gets closed automatically
with open("data.txt", "r") as file:
    content = file.read()
# The file is already closed
```

How it works:

- `__enter__()` — called upon entering the `with` block. Its return value is assigned to the variable after `as`.
- `__exit__()` — called upon exiting the `with` block, **even if an exception was raised**.

Creating a custom context manager:

```
class Timer:
    def __enter__(self):
        import time
        self.start = time.time()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        import time
        elapsed = time.time() - self.start
        print(f"Execution time: {elapsed:.2f} sec")
        return False  # Don't suppress exceptions

with Timer():
    total = sum(range(1_000_000))
# Execution time: 0.03 sec
```

Where context managers are used:

- Working with **files** (`open`)
- **Locks** in multithreading (`threading.Lock`)
- **Database connections** (automatic commit/rollback)
- **Temporary resources** (temporary files, network connections)

Question 22

How to read and write files in Python?

Opening a file:

The `open()` function takes the path to the file and a mode:

- `'r'` — read (default)
- `'w'` — write (overwrites the file)
- `'a'` — append (adds to the end of the file)
- `'rb'` / `'wb'` — read / write in binary mode

Reading a file:

```
# Read the entire file
with open("data.txt", "r", encoding="utf-8") as f:
    content = f.read()

# Read line by line
with open("data.txt", "r", encoding="utf-8") as f:
    for line in f:
        print(line.strip())

# Read all lines into a list
with open("data.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()
```

Writing to a file:

```
# Overwrite the file
with open("output.txt", "w", encoding="utf-8") as f:
    f.write("First line\n")
    f.write("Second line\n")

# Append to the end
with open("output.txt", "a", encoding="utf-8") as f:
    f.write("Another line\n")
```

Encodings:

Always specify `encoding="utf-8"` to avoid issues with special characters and non-Latin alphabets:

```
# Without specifying encoding, a UnicodeDecodeError might occur
with open("data.txt", "r", encoding="utf-8") as f:
    content = f.read()
```

Important:

Always use the `with` statement — it ensures the file is properly closed even if an error occurs.

Question 23

What are classes and objects in Python?

> *A **class** is a template (blueprint) for creating objects. An **object** is a specific instance of a class.*

Defining a class:

```
class Dog:
    # Class attribute (shared by all instances)
    species = "Canis familiaris"

    # Constructor - gets called when creating an object
    def __init__(self, name, age):
        # Instance attributes (unique to each object)
        self.name = name
        self.age = age

    # Instance method
    def bark(self):
        return f"{self.name} says: Woof!"
```

Creating objects:

```
dog1 = Dog("Bobik", 3)
dog2 = Dog("Sharik", 5)

print(dog1.name)      # Bobik
print(dog2.bark())    # Sharik says: Woof!
print(dog1.species)   # Canis familiaris
```

self:

- `self` is a reference to the current instance of the class.
- It is passed automatically when a method is called.
- Through `self`, you can access the object's attributes and methods.

Class attributes vs instance attributes:

```
class Counter:
    count = 0  # Class attribute - shared by all

    def __init__(self):
        Counter.count += 1  # Modify the class attribute
        self.id = Counter.count  # Instance attribute

c1 = Counter()
c2 = Counter()
print(Counter.count)  # 2
print(c1.id, c2.id)   # 1 2
```

Question 24

What is inheritance and how does it work in Python?

> ***Inheritance*** *is an OOP mechanism where a child class inherits attributes and methods from a parent class.*

Basic inheritance:

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

    def speak(self):
        return f"{self.name} makes a sound"

class Dog(Animal):
    def speak(self):
        return f"{self.name} says: Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says: Meow!"

dog = Dog("Bobik")
print(dog.speak())  # Bobik says: Woof!
```

super() — calling a parent method:

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

class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, age)  # Call parent's __init__
        self.breed = breed

dog = Dog("Bobik", 3, "Labrador")
print(dog.name, dog.breed)  # Bobik Labrador
```

Checking inheritance:

```
print(isinstance(dog, Dog))     # True
print(isinstance(dog, Animal))  # True
print(issubclass(Dog, Animal))  # True
```

Overriding methods:

A child class can replace or extend a parent's method:

```
class Shape:
    def area(self):
        return 0

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

rect = Rectangle(5, 3)
print(rect.area())  # 15
```

Multiple inheritance:

Python supports inheriting from multiple classes. The method lookup order is determined by the MRO (Method Resolution Order) algorithm, which can be inspected via `ClassName.mro()`.

Question 25

What is encapsulation in Python?

> ***Encapsulation*** *is an OOP principle where an object's internal data is hidden from direct outside access.*

Naming conventions:

In Python, there are no strict access modifiers (`private` , `public`). Instead, conventions are used:

- `name` — public attribute, accessible to everyone.
- `_name` — "protected" (by convention), intended for internal use only.
- `__name` — "private", Python applies **name mangling** (changing the name).

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

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
# print(account.__balance)  # AttributeError
print(account.get_balance())  # 1000

# Name mangling - the attribute is accessible via the modified name
print(account._BankAccount__balance)  # 1000
```

@property — controlled access:

Allows using methods as if they were attributes:

```
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Below absolute zero!")
        self._celsius = value

    @property
    def fahrenheit(self):
        return self._celsius * 9 / 5 + 32

temp = Temperature(25)
print(temp.celsius)     # 25
print(temp.fahrenheit)  # 77.0
temp.celsius = 30       # Uses the setter
```

Question 26

What is polymorphism in Python?

> ***Polymorphism*** *is the ability of objects from different classes to respond differently to the same method call.*

Polymorphism through method overriding:

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

class Dog:
    def speak(self):
        return "Woof!"

class Duck:
    def speak(self):
        return "Quack!"

# Same interface, different behavior
animals = [Cat(), Dog(), Duck()]
for animal in animals:
    print(animal.speak())
```

Duck Typing:

> *"If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."*

Python does not check the object's type — what matters is the presence of the required method:

```
class File:
    def read(self):
        return "data from a file"

class Database:
    def read(self):
        return "data from a DB"

def load_data(source):
    # The type doesn't matter, only that it has a 'read' method
    return source.read()

print(load_data(File()))      # data from a file
print(load_data(Database()))  # data from a DB
```

Polymorphism of built-in functions:

```
# len() works with different types
print(len("Python"))    # 6
print(len([1, 2, 3]))   # 3
print(len({"a": 1}))    # 1

# + behaves differently
print(1 + 2)           # 3 (addition)
print("Hello, " + "world!")  # Hello, world! (concatenation)
```

Question 27

What are static and class methods?

Regular method (instance method):

Receives a reference to the instance (`self`) as its first argument:

```
class MyClass:
    def instance_method(self):
        return f"Called for {self}"
```

@classmethod — a method of the class:

Receives a reference to the **class** (`cls`) instead of an instance:

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

    @classmethod
    def from_string(cls, data_string):
        name, age = data_string.split(",")
        return cls(name, int(age))

# Alternative constructor
user = User.from_string("Anna,25")
print(user.name)  # Anna
```

@staticmethod — a static method:

Receives neither `self` nor `cls` . It behaves like a regular function placed inside a class namespace:

```
class MathUtils:
    @staticmethod
    def is_even(n):
        return n % 2 == 0

print(MathUtils.is_even(4))  # True
```

When to use which:

- **Instance method** — when you need access to the object's attributes (`self`).

- **@classmethod** — for alternative constructors or when modifying class attributes.
- **@staticmethod** — for utility functions that logically belong to the class but do not need access to the instance or the class.

Question 28

What are magic (dunder) methods?

> ***Magic methods*** *(dunder methods, short for "double underscore") are special methods surrounded by double underscores that define the behavior of objects during built-in operations.*

String representation:

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

    def __str__(self):
        # For the end user (print, str())
        return f"{self.name}: ${self.price}"

    def __repr__(self):
        # For the developer (debugging, repr())
        return f"Product('{self.name}', {self.price})"

p = Product("Book", 50)
print(p)       # Book: $50
print(repr(p)) # Product('Book', 50)
```

Object comparison:

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

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __lt__(self, other):
        return (self.x ** 2 + self.y ** 2) < (other.x ** 2 + other.y ** 2)

Point(1, 2) == Point(1, 2)  # True
Point(1, 2) < Point(3, 4)   # True
```

Other useful magic methods:

- `__len__` — behavior for `len(obj)`
- `__getitem__` — index access `obj[key]`
- `__contains__` — the `in` operator
- `__add__` — the `+` operator
- `__call__` — calling an object like a function `obj()`

```
class Basket:
    def __init__(self):
        self.items = []

    def __len__(self):
        return len(self.items)

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

basket = Basket()
basket.items.append("apple")
print(len(basket))           # 1
print("apple" in basket)     # True
```

Question 29

What are decorators and how do they work?

> *A **decorator** is a function that takes another function and returns a new one, extending its behavior without modifying the original code.*

A simple decorator:

```
def log_call(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Result: {result}")
        return result
    return wrapper

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

add(3, 5)
# Calling function: add
# Result: 8
```

How it works under the hood:

```
# Using @decorator syntax
@log_call
def add(a, b):
    return a + b

# Is equivalent to:
def add(a, b):
    return a + b
add = log_call(add)
```

Preserving metadata (@wraps):

```python
from functools import wraps

def log_call(func):
    @wraps(func)  # Keeps the original name and docstring
    def wrapper(*args, **kwargs):
        print(f"Calling: {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@log_call
def add(a, b):
    """Adds two numbers."""
    return a + b

print(add.__name__)  # add (without @wraps it would be 'wrapper')
print(add.__doc__)   # Adds two numbers.
```

Practical examples of decorators:

- Logging function calls
- Measuring execution time
- Caching results
- Access control/authentication
- Input validation

Question 30

What are abstract classes and why are they needed?

> *An **abstract class** is a class that cannot be instantiated directly. It defines an interface (a set of mandatory methods) for its child classes.*

The abc module:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        """Calculate the area of the shape"""
        pass

    @abstractmethod
    def perimeter(self):
        """Calculate the perimeter of the shape"""
        pass

# shape = Shape()  # TypeError: Can't instantiate abstract class Create
```

Implementing an abstract class:

```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

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

    def area(self):
        import math
        return math.pi * self.radius ** 2

    def perimeter(self):
        import math
        return 2 * math.pi * self.radius

rect = Rectangle(5, 3)
print(rect.area())       # 15
print(rect.perimeter())  # 16
```

Why they are needed:

- **Contract**: they guarantee that any child class implements all the required methods.
- **Documentation**: they clearly show what methods must be implemented.
- **Early error detection**: if an abstract method is missing in the child class, Python will raise a `TypeError` at instantiation, rather than failing later when the method is called.

Question 31

How does importing modules work in Python?

> *A **module** is a file with a `.py` extension containing Python code (functions, classes, variables). Modules help organize and reuse code.*

Ways to import:

```
# Importing an entire module
import math
print(math.sqrt(16))  # 4.0

# Importing specific objects
from math import sqrt, pi
print(sqrt(16))  # 4.0

# Importing with an alias
import datetime as dt
now = dt.datetime.now()
```

Creating your own module:

```
# utils.py
def greet(name):
    return f"Hello, {name}!"

# main.py
from utils import greet
print(greet("World"))
```

__name__ == "__main__":

Allows distinguishing whether the file is run directly or imported as a module:

```
# my_module.py
def main():
    print("Running the module directly")

if __name__ == "__main__":
    # This code executes only when running directly:
    # python my_module.py
    main()
```

Packages:

A package is a directory with an `__init__.py` file containing multiple modules:

```
my_package/
    __init__.py
    module_a.py
    module_b.py
```

```
from my_package.module_a import some_function
```

Question 32

What is a virtual environment and why is it needed?

> *A **virtual environment** is an isolated Python environment where packages are installed only for a specific project, without affecting the global installation.*

Why it's needed:

- Different projects might require **different versions** of the same library.
- To avoid **conflicts** between dependencies of different projects.
- To lock exact package versions for **reproducibility**.

Creation and usage (venv):

```
# Create a virtual environment
python -m venv venv

# Activation
# macOS/Linux:
source venv/bin/activate
# Windows:
venv\Scripts\activate

# Deactivation
deactivate
```

Package management (pip):

```
# Install a package
pip install requests

# Install a specific version
pip install requests==2.31.0

# List installed packages
pip list

# Save dependencies to a file
pip freeze > requirements.txt

# Install dependencies from a file
pip install -r requirements.txt
```

requirements.txt:

```
requests==2.31.0
flask==3.0.0
pytest==7.4.3
```

This file locks all project dependencies, allowing you to recreate the environment on another machine.

Question 33

How to work with JSON in Python?

> ***JSON*** *(JavaScript Object Notation) is a text-based format for data exchange. Python provides a built-in* `json` *module for working with it.*

Main functions:

- `json.dumps()` — Python object → JSON string
- `json.loads()` — JSON string → Python object
- `json.dump()` — Python object → JSON file
- `json.load()` — JSON file → Python object

Serialization (Python → JSON):

```
import json

data = {
    "name": "Anna",
    "age": 25,
    "hobbies": ["reading", "swimming"],
    "active": True
}

# To a string
json_string = json.dumps(data, ensure_ascii=False, indent=2)
print(json_string)

# To a file
with open("data.json", "w", encoding="utf-8") as f:
    json.dump(data, f, ensure_ascii=False, indent=2)
```

Deserialization (JSON → Python):

```
# From a string
json_string = '{"name": "Anna", "age": 25}'
data = json.loads(json_string)
print(data["name"])  # Anna

# From a file
with open("data.json", "r", encoding="utf-8") as f:
    data = json.load(f)
```

Type mapping:

JSON	Python
object	dict
array	list
string	str
number	int/float
true/false	True/False
null	None

Question 34

What are type annotations in Python?

> ***Type annotations*** *are a way to declare the expected types for function arguments, return values, and variables. They do* ***not affect*** *the runtime behavior of the program but serve as helpful aids during development.*

Basic syntax:

```
def greet(name: str) -> str:
    return f"Hello, {name}!"

age: int = 25
price: float = 19.99
is_active: bool = True
```

Collections:

```
# Python 3.9+
def process(items: list[str]) -> dict[str, int]:
    return {item: len(item) for item in items}

# Nested types
matrix: list[list[int]] = [[1, 2], [3, 4]]
```

The typing module:

```
from typing import Optional, Union

# Can be str or None
def find_user(user_id: int) -> Optional[str]:
    if user_id == 1:
        return "Anna"
    return None

# Can be int or str
def parse(value: Union[int, str]) -> str:
    return str(value)
```

Why use annotations:

- **Documentation**: the expected types are clear without reading the function body.
- **IDEs**: autocomplete and code suggestions work better.
- **Static analysis**: tools like `mypy` can catch type errors prior to running code.
- **Collaboration**: makes the codebase easier for other developers to understand.

Important note:

Annotations are merely **hints**, not strict enforcements. Python **does not** enforce types at runtime:

```
def add(a: int, b: int) -> int:
    return a + b

add("hello", " world")  # Will run without error yielding: "hello world"
```

Question 35

What is the difference between a shallow copy and a deep copy?

Shallow copy:

Creates a new object but **does not copy nested objects** — it maintains references to the originals:

```
import copy

original = [[1, 2, 3], [4, 5, 6]]
shallow = copy.copy(original)

shallow[0][0] = 999
print(original[0][0])  # 999 – the original has changed too!
```

Deep copy:

Creates a **completely independent clone**, including all nested objects:

```
import copy

original = [[1, 2, 3], [4, 5, 6]]
deep = copy.deepcopy(original)

deep[0][0] = 999
print(original[0][0])  # 1 – the original remains unchanged
```

Ways to make a shallow copy:

```
# For lists
lst = [1, 2, 3]
copy1 = lst.copy()
copy2 = lst[:]
copy3 = list(lst)

# For dictionaries
d = {"a": 1}
copy4 = d.copy()
copy5 = dict(d)

# Universal
import copy
copy6 = copy.copy(lst)
```

When to use which:

- **Shallow** — when the collection only contains immutable elements (numbers, strings).
- **Deep** — when there are nested mutable objects (lists within lists, dictionaries within lists).

```
# Shallow is sufficient
nums = [1, 2, 3]  # Elements are immutable ints
safe_copy = nums.copy()

# Deep copy is required
matrix = [[1, 2], [3, 4]]  # Nested lists
safe_copy = copy.deepcopy(matrix)
```

Question 36

What are iterators and the iteration protocol?

> *An **iterator** is an object that returns elements one by one using the `__next__()` method and signals completion by raising a `StopIteration` exception.*

The iteration protocol:

- `__iter__()` — returns the iterator object itself.
- `__next__()` — returns the next element or raises a `StopIteration` exception.

How a for loop works:

```
# What 'for' does under the hood
nums = [1, 2, 3]

# for num in nums:
#     print(num)

# Equivalent:
iterator = iter(nums)       # Calls nums.__iter__()
while True:
    try:
        num = next(iterator)  # Calls iterator.__next__()
        print(num)
    except StopIteration:
        break
```

Creating a custom iterator:

```
class Countdown:
    def __init__(self, start):
        self.current = start

    def __iter__(self):
        return self

    def __next__(self):
        if self.current <= 0:
            raise StopIteration
        value = self.current
        self.current -= 1
        return value

for num in Countdown(5):
    print(num)  # 5, 4, 3, 2, 1
```

Built-in iterables:

- `list`, `tuple`, `str`, `dict`, `set` are **iterable** (they have an `__iter__` method).
- `range()`, `map()`, `filter()`, `zip()` return **iterators**.
- An **iterator** can only be traversed **once**, while an **iterable object** can be looped over multiple times.

In practice:

The iteration protocol underpins `for` loops, generators, and many built-in functions. Custom iterators are useful when you need to process data in chunks — for example, reading a large file line by line instead of loading it all into memory.

Question 37

What is the difference between is and ==?

== (value comparison):

Checks whether the **values match** between two objects:

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

print(a == b)  # True – values are the same
```

is (identity comparison):

Checks whether two variables point to the **exact same object** in memory:

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

print(a is b)  # False – they are different objects
print(id(a), id(b))  # Different memory addresses

c = a
print(a is c)  # True – c points to the same object
```

Caching of small integers:

Python caches integers from **-5 to 256**, so:

```
x = 100
y = 100
print(x is y)  # True – cached object

x = 1000
y = 1000
print(x is y)  # False – different objects
```

Rules of usage:

- Use `is` only for comparison against `None`:

```
# Correct
if value is None:
    print("No value")

# Incorrect
if value == None:
    print("No value")
```

- When comparing values, **always** use `==`.
- `is` only checks the `id()` of the objects — this is useful mostly for singletons like `None`, `True`, or `False`.

Question 38

How does memory management work in Python?

> *Python automatically handles memory management, relieving developers of manual memory allocation and deallocation.*

Reference Counting:

Every object keeps track of its **reference count** — the number of variables pointing to it. When the count reaches **zero**, the object is deleted:

```
import sys

a = [1, 2, 3]
print(sys.getrefcount(a))  # 2 (a + the function parameter itself)

b = a  # Another reference
print(sys.getrefcount(a))  # 3

del b  # Remove a reference
print(sys.getrefcount(a))  # 2
```

Garbage Collector:

Reference counting fails when there are **circular references**:

```
# Circular reference
a = []
b = []
a.append(b)
b.append(a)
del a, b
# The reference count won't reach zero, but the objects are unreachable
```

To solve this, Python runs a **Garbage Collector** (the `gc` module) that discovers and cleans up such cycles.

Key takeaways:

- **Reference counting** is the primary mechanism and runs immediately.
- **Garbage collector** is a secondary mechanism designed to break circular references.
- **del** removes a reference to an object, not necessarily the object itself.

- Developers rarely need to get involved in memory management — Python handles it perfectly.

Question 39

What is a closure in Python?

> *A **closure** is a nested function that "remembers" variables from the enclosing function's scope, even after the enclosing function has finished its execution.*

How it works:

```
def make_multiplier(factor):
    def multiply(number):
        return number * factor  # 'factor' is captured from the outer function
    return multiply

double = make_multiplier(2)
triple = make_multiplier(3)

print(double(5))   # 10
print(triple(5))   # 15
```

Conditions for a closure:

- There must be a **nested function**.
- The nested function must refer to an environment variable from the **enclosing function**.
- The enclosing function must **return** the nested function.

Practical applications:

```
# Counter
def make_counter(start=0):
    count = start
    def counter():
        nonlocal count
        count += 1
        return count
    return counter

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

# Logger
def make_logger(prefix):
    def log(message):
        print(f"[{prefix}] {message}")
    return log

error_log = make_logger("ERROR")
error_log("File not found")  # [ERROR] File not found
```

Question 40

What is the difference between threading and multiprocessing in Python?

Multithreading (threading):

Threads operate within a **single process** and share the same memory space:

```
import threading

def download(url):
    print(f"Downloading {url}")

threads = []
for url in ["url1", "url2", "url3"]:
    t = threading.Thread(target=download, args=(url,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()  # Wait for all threads to finish
```

Multiprocessing (multiprocessing):

Each process gets its **own memory space** and its own **Python interpreter**:

```
from multiprocessing import Process

def heavy_computation(n):
    return sum(i * i for i in range(n))

processes = []
for n in [10_000_000, 20_000_000]:
    p = Process(target=heavy_computation, args=(n,))
    processes.append(p)
    p.start()

for p in processes:
    p.join()
```

The GIL (Global Interpreter Lock):

The GIL is a CPython mechanism that allows **only one thread** to execute Python bytecode at a time. As a result:

- Threads **do not speed up** tasks performing heavy calculations (CPU-bound tasks).
- Threads **do speed up** tasks waiting on external events (I/O-bound tasks): network requests, reading files.

When to use which:

- **threading** — for I/O-bound tasks: file downloads, API calls, database queries.
- **multiprocessing** — for CPU-bound tasks: data processing, complex mathematical calculations, and algorithms.

Feature	threading	multiprocessing
Memory	Shared	Separate
GIL impact	Constrained	Unaffected
Overhead	Low	High
Best used for	I/O-bound	CPU-bound