.

1. Basics of Python

1. What are Python’s key features?

Python's key features include:

- **Easy to Learn and Use**: Python has a simple, clean syntax that's very readable.
- **Interpreted Language**: Code is executed line by line, which makes debugging easier.
- **High-Level Language**: You don't need to worry about low-level details like memory management.
- **Dynamically Typed**: You don't need to declare variable types. The interpreter infers them at runtime.
- **Platform Independent**: Python code can run on different operating systems (Windows, macOS, Linux) without changes.
- **Extensive Standard Library**: Python comes with a massive collection of modules for various tasks, from web development to data science.
- **Object-Oriented**: Python fully supports object-oriented programming (OOP) principles like encapsulation, inheritance, and polymorphism.

2. Difference between Python 2 and Python 3?

The main differences between Python 2 and 3 are:

- **`print` Statement**: Python 2 used `print` as a statement (`print "Hello"`), while Python 3 uses it as a function (`print("Hello")`).
- **Unicode Support**: Python 3 handles text strings as Unicode by default, which simplifies working with different languages and characters. Python 2 used ASCII strings by default.
- **Integer Division**: In Python 2, `5 / 2` resulted in `2` (integer division). In Python 3, it results in `2.5` (float division). To get integer division in Python 3, you'd use `//`.
- **`xrange` vs. `range`**: In Python 2, `range()` created a full list in memory, while `xrange()` created a generator-like object. In Python 3, `range()` behaves like Python 2's `xrange()`, returning an iterator, which is more memory efficient.

3. What are Python data types?

Python has several built-in data types:

- **Numeric Types**: `int` (integers), `float` (floating-point numbers), `complex` (complex numbers).
    - `x = 10` (int)
    - `y = 10.5` (float)
- **Boolean Type**: `bool` (True or False).
    - `is_true = True`
- **Sequence Types**: `str` (strings), `list` (lists), `tuple` (tuples), `range`.
    - `s = "hello"`
    - `l = [1, 2, 3]`

    - t = (1, 2, 3)
- **Set Types**: set (unordered collection of unique items), frozenset.
    - s = {1, 2, 3}
- **Mapping Type**: dict (dictionaries).
    - d = {"key": "value"}
- **None Type**: NoneType (a special type for None).
    - n = None

4. Explain mutable vs immutable objects in Python.

- **Mutable Objects**: Their state can be changed after they're created. Examples include **lists, dictionaries, and sets**. When you modify a mutable object, its ID (memory address) remains the same.
- Python

```
my_list = [1, 2, 3]
print(id(my_list))
my_list.append(4)
print(id(my_list)) # ID remains the same
```

-
-
- **Immutable Objects**: Their state cannot be changed after creation. Examples include **numbers, strings, and tuples**. If you try to "modify" an immutable object, you're actually creating a new object in memory with a different ID.
- Python

```
my_str = "hello"
print(id(my_str))
my_str = my_str + " world"
print(id(my_str)) # A new ID is created
```

-
-

5. What are Python’s built-in data structures (list, tuple, set, dict)?

- **List**: An ordered, mutable collection of items. Lists can contain items of different data types. They are defined by square brackets [].
    - my_list = [1, "hello", 3.14]
- **Tuple**: An ordered, immutable collection of items. Tuples are faster than lists and are used for data that shouldn't change. They are defined by parentheses ().
    - my_tuple = (1, "hello", 3.14)
- **Set**: An unordered collection of unique, mutable items. Sets are useful for tasks like removing duplicates or performing set operations (union, intersection). They are defined by curly braces {}.
    - my_set = {1, 2, 3, 2} # result will be {1, 2, 3}

- **Dictionary**: An unordered, mutable collection of key-value pairs. Dictionaries are optimized for retrieving values when the key is known. They are defined by curly braces {}.
    - `my_dict = {"name": "Alice", "age": 30}`

6. Difference between list and tuple?

- **Mutability**: **Lists are mutable**, meaning you can add, remove, or change elements. **Tuples are immutable**, and their elements cannot be changed after creation.
- **Syntax**: Lists use square brackets `[]`, while tuples use parentheses `()`.
- **Performance**: Tuples are generally faster than lists because of their immutable nature.
- **Use Case**: Use lists for data that needs to be modified, and tuples for data that should remain constant, like coordinates or database records.

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

This question is a repeat of question 35. See the answer for question 35 below.

8. What is Python's memory management mechanism?

Python uses a private heap space to manage memory. All objects and data structures reside in this heap. The Python memory manager handles the allocation and deallocation of this memory. It uses a combination of techniques:

- **Reference Counting**: Python keeps a count of how many references point to an object. When the count drops to zero, the object is a candidate for garbage collection.
- **Garbage Collection**: For cyclical references (where objects reference each other but are no longer accessible from the program), Python's garbage collector, which is a generational collector, cleans up these unused objects.
- **Small Object Caching**: Python caches small integers and strings to improve performance and memory usage.

9. What is `__init__` in Python?

`__init__` is a special method, often called a **constructor**, that gets automatically called when you create a new instance of a class. It's used to initialize the object's attributes with the values passed to the constructor.

Example:

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

```
my_dog = Dog("Buddy", 5)
print(my_dog.name) # Buddy
```

10. What is Python's garbage collection mechanism?

Python's garbage collector (GC) is a system for reclaiming memory that is no longer in use. It primarily uses **reference counting**, where each object has a count of how many variables refer to it. When this count becomes zero, the object's memory is freed.

However, reference counting can't handle circular references (e.g., Object A refers to B, and B refers to A, but no other variables refer to either). To solve this, Python's GC has a **cyclic garbage collector** that periodically runs to detect and clean up these cycles. The cyclic GC uses a generational approach, dividing objects into generations (0, 1, 2) based on how long they've been alive. Newer objects (Generation 0) are checked more frequently, which is more efficient.

---

2. Python Functions & OOP

11. What are Python functions? How are they defined?

A function is a block of reusable code that performs a specific task. They help break down a large program into smaller, manageable, and reusable parts, improving code organization and reducing redundancy.

Functions are defined using the `def` keyword, followed by the function name, parentheses `()`, and a colon `:`. The code block is indented.

Example:

```
Python

def greet(name):
    """This function greets the person passed in as an argument."""
    return f"Hello, {name}!"

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

12. What are `*args` and `**kwargs`?

These are special syntax used in function definitions to handle a variable number of arguments.

- `*args`: Allows a function to accept a variable number of non-keyword (positional) arguments. It collects these arguments into a **tuple**. The name `args` is a convention; you can use any name.
- ```
  Python
  ```
  ```

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

print(sum_all(1, 2, 3)) # 6
```

-
-
- **kwargs: Allows a function to accept a variable number of keyword (named) arguments. It collects these arguments into a **dictionary**. The name kwargs is a convention.
- Python

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=30)
# name: Alice
# age: 30
```

-
-

13. Explain lambda functions in Python.

A lambda function is a small, anonymous function defined with the lambda keyword. It can have any number of arguments but can only have one expression. The result of this expression is the return value. Lambda functions are often used for short, simple operations and are typically used as arguments to higher-order functions like map(), filter(), and sorted().

Syntax: lambda arguments: expression

Example:

Python

```
add_one = lambda x: x + 1
print(add_one(5)) # 6

# Using with a higher-order function
my_list = [1, 5, 2, 8]
sorted_list = sorted(my_list, key=lambda x: -x)
print(sorted_list) # [8, 5, 2, 1] (sorts in descending order)
```

14. Difference between @staticmethod, @classmethod, and instance methods.

This is a key concept in Python OOP:

- **Instance Methods**: The most common type of method. It takes `self` as the first argument, which refers to the specific instance of the class. It can access and modify both instance-level and class-level attributes.
- Python

```
class MyClass:
    def instance_method(self):
        print("This is an instance method.")
```

-
-
- **@classmethod**: A method that belongs to the class itself, not a specific instance. It takes `cls` as the first argument, which refers to the class. It's often used to create factory methods that return an instance of the class.
- Python

```
class MyClass:
    @classmethod
    def class_method(cls):
        print("This is a class method.")
```

-
-
- **@staticmethod**: A method that belongs to the class but doesn't take `self` or `cls` as an argument. It's essentially a regular function that is logically part of the class but doesn't interact with the class or its instances. It's often used for utility or helper functions.
- Python

```
class MyClass:
    @staticmethod
    def static_method():
        print("This is a static method.")
```

-
-

15. What are decorators in Python?

A decorator is a design pattern in Python that allows you to add new functionality to an existing function or class without modifying its structure. It's a function that takes another function as an argument, adds some new behavior, and returns the modified function. Decorators are typically used for things like logging, timing, or authentication.

Example:

Python

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
# Something is happening before the function is called.
# Hello!
# Something is happening after the function is called.
```

16. What is the difference between inheritance and composition?

- **Inheritance** (**"is-a"** relationship): A mechanism where a new class (subclass) inherits attributes and methods from an existing class (superclass). It promotes code reuse but can lead to a rigid class hierarchy.
    - **Example**: A Dog **is a** Animal.
- Python

```
class Animal:
    def speak(self): pass
class Dog(Animal):
    def speak(self): print("Woof!")
```

-
-
- **Composition** (**"has-a"** relationship): A design pattern where a class contains an object of another class as an attribute. It promotes flexibility and loose coupling.
    - **Example**: A Car **has a** Engine.
- Python

```
class Engine:
    def start(self): print("Engine started.")
class Car:
    def __init__(self):
        self.engine = Engine() # Composition
    def start(self):
        self.engine.start()
```

-
-

Key difference: Inheritance models a hierarchy, while composition models a relationship of parts to a whole. Composition is generally preferred over inheritance because it offers more flexibility.

17. Explain multiple inheritance in Python.

Multiple inheritance is a feature where a class can inherit from multiple parent classes. This allows a subclass to combine the functionalities of all its parent classes. While powerful, it can lead to complex and ambiguous situations, such as the "diamond problem," where a method is defined in multiple parent classes.

Example:

```
Python

class A:
    def method(self):
        print("Method from A")

class B:
    def method(self):
        print("Method from B")

class C(A, B):
    pass

c = C()
c.method() # Method from A (due to Method Resolution Order)
```

18. What is method resolution order (MRO)?

Method Resolution Order (MRO) is the order in which Python searches for a method in a class hierarchy, especially in the case of multiple inheritance. Python uses the **C3 linearization algorithm** to determine the MRO. You can check the MRO of a class using the `mro()` method or the `__mro__` attribute.

Example:

```
Python

class A: pass
class B(A): pass
class C(B): pass

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

19. What is the difference between `is` and `==`?

- `==` checks for **value equality**. It compares the values of two objects.
- Python

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

-
-
- `is` checks for **identity equality**. It compares the memory addresses (`id()`) of two objects. It's used to determine if two variables point to the exact same object in memory.
- Python

```
a = [1, 2, 3]
b = [1, 2, 3]
print(a is b) # False (they are different objects)

c = a
print(a is c) # True (they point to the same object)
```

-
-

20. What is duck typing in Python?

Duck typing is a programming concept where the type of an object is less important than the methods it defines. The core idea is: "If it walks like a duck and quacks like a duck, then it must be a duck." In Python, if an object has the required methods and attributes, it can be used in a particular context, regardless of its formal type.

Example:

Python

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

class Cat:
    def speak(self):
        print("Meow!")

def make_it_speak(animal):
    animal.speak()

make_it_speak(Dog()) # Woof!
make_it_speak(Cat()) # Meow!
```

Here, make_it_speak works with both Dog and Cat because they both have a speak() method. Their formal type doesn't matter.

---

3. Python Modules & Packages

21. Difference between module and package?

- **Module**: A single Python file (.py) containing Python code, such as functions, classes, and variables.
- **Package**: A directory that contains multiple Python modules and an __init__.py file (even if it's empty). Packages are a way to organize related modules into a directory hierarchy.

Analogy: A module is like a single book, while a package is like a bookshelf that holds multiple related books (modules).

22. How does Python import work?

When you use an import statement, Python follows these steps:

1. **Check sys.modules**: It first checks if the module is already loaded in sys.modules, which is a dictionary of loaded modules. If it's there, Python uses the existing module object.
2. **Find the module**: If not, Python searches for the module file in the directories listed in sys.path. This includes the current directory, the PYTHONPATH environment variable, and the standard library directories.
3. **Execute the module**: Once found, Python executes the code in the module file. This creates a new module object.
4. **Add to sys.modules**: The new module object is then added to sys.modules, so it can be reused later without re-execution.

23. What is __name__ == "__main__" in Python?

The __name__ variable is a special built-in variable that holds the name of the current module.

- When a Python script is run directly, __name__ is set to "__main__".
- When the script is imported as a module into another script, __name__ is set to the name of the module.

The if __name__ == "__main__": block is a common idiom used to ensure that a specific block of code only runs when the script is executed directly, not when it's imported.

Example:

Python

```
# my_module.py
def main():
```

```
    print("This is the main function.")

if __name__ == "__main__":
    main() # This will only run if my_module.py is executed directly
```

24. Explain virtual environments in Python.

A **virtual environment** is a self-contained directory that holds a specific Python interpreter and its own set of installed packages. It provides an isolated environment for each Python project. This prevents conflicts between different projects that may require different versions of the same library.

How to create and use one:

```
Bash

# Create a virtual environment named 'my_env'
python3 -m venv my_env

# Activate it (on macOS/Linux)
source my_env/bin/activate

# Install packages inside the env
pip install requests

# Deactivate it
deactivate
```

25. What is pip and how does it work?

pip (Package Installer for Python) is the standard package manager for Python. It's used to install, uninstall, and manage Python packages from the Python Package Index (PyPI) or other repositories.

How it works:

1. You run a command like `pip install package_name`.
2. `pip` sends a request to PyPI (or a specified repository) to find the package.
3. PyPI returns the necessary information, including the URL to the package files.
4. `pip` downloads the package files (usually `.whl` or `.tar.gz`) from the repository.
5. It then unpacks the files and installs them into your Python environment's `site-packages` directory. It also handles dependencies, automatically installing any other packages the requested package needs.

---

4. Exception Handling

26. How does exception handling work in Python?

Exception handling is a mechanism to deal with errors that occur during the execution of a program, preventing the program from crashing. It's done using `try`, `except`, `else`, and `finally` blocks.

- The `try` block contains the code that might raise an exception.
- The `except` block catches a specific type of exception and handles it. You can have multiple `except` blocks for different exceptions.
- The `else` block (optional) runs only if the `try` block completes successfully without raising an exception.
- The `finally` block (optional) always runs, regardless of whether an exception occurred or not. It's typically used for cleanup operations like closing files.

Example:

```
Python

try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("That's not a valid number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print(f"Result: {result}")
finally:
    print("Execution complete.")
```

27. Difference between `try/except/finally` and `try/except/else`.

- **`try/except/finally`**: The `finally` block **always** executes, whether an exception was raised and handled, or not. This is useful for cleanup tasks that must happen regardless of the outcome (e.g., closing a file or a database connection).
- **`try/except/else`**: The `else` block executes **only if** the `try` block runs without any exceptions. This is useful for code that should only run on success, separating it from the code that might raise an exception.

28. How to create custom exceptions in Python?

You can create your own custom exceptions by creating a new class that inherits from the built-in `Exception` class or one of its subclasses. This allows you to define more specific and descriptive error types for your applications.

Example:

```
Python

class InsufficientFundsError(Exception):
    """Custom exception for bank account operations."""
```

```
    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
        super().__init__(f"Attempt to withdraw {amount} with only {balance} in account.")

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

try:
    withdraw(100, 200)
except InsufficientFundsError as e:
    print(e)
    # Output: Attempt to withdraw 200 with only 100 in account.
```

29. What is the difference between errors and exceptions?

- **Errors**: These are serious problems that your program can't recover from, and they often prevent the program from running at all. Examples include **syntax errors** (e.g., a typo in your code) or **IndentationErrors**.
- **Exceptions**: These are events that occur during the execution of a program that disrupt the normal flow. They can be handled and caught by your code using `try-except` blocks. Examples include `ValueError`, `TypeError`, `ZeroDivisionError`, and `FileNotFoundError`.

In short, **errors are fatal problems**, while **exceptions are problems that can be handled** by your code.

---

5. Advanced Python Concepts

30. What are Python generators? How do they work with `yield`?

A **generator** is a special type of function that returns an iterator. It's a memory-efficient way to create iterables, especially for large datasets. Instead of creating and returning an entire list at once, a generator **yields** one item at a time.

- The `yield` keyword pauses the function's execution and returns a value to the caller.
- The function's state is saved, so the next time `next()` is called on the generator, it resumes execution from where it left off.

Example:

```
Python

def fibonacci(limit):
    a, b = 0, 1
    while a < limit:
```

```
        yield a
        a, b = b, a + b

# The generator object is created
fib_gen = fibonacci(10)

# We can iterate over it
print(next(fib_gen)) # 0
print(next(fib_gen)) # 1
print(next(fib_gen)) # 1
print(next(fib_gen)) # 2
print(next(fib_gen)) # 3
```

31. What are Python iterators and iterables?

- **Iterable**: An object that can be looped over (e.g., a list, tuple, string, dictionary). It has an `__iter__` method that returns an iterator.
    - `my_list = [1, 2, 3]` is an iterable.
- **Iterator**: An object that represents a stream of data and keeps track of its current position. It has a `__next__` method that returns the next item in the stream. When there are no more items, it raises a `StopIteration` exception.
    - `my_iterator = iter(my_list)` is an iterator.

Analogy: An iterable is like a movie (the whole collection of frames), while an iterator is like a movie projector that shows one frame at a time.

32. Difference between generator and iterator?

- **Generators are a way to create iterators.** When a function contains the `yield` keyword, it becomes a generator function and returns a generator object, which is an iterator.
- **Iterators are an object protocol.** Any object that has the `__iter__` and `__next__` methods is an iterator. Generators are simply a convenient, memory-efficient way to implement this protocol.

33. What are Python comprehensions (list, dict, set)?

Comprehensions provide a concise and elegant way to create new sequences (lists, dictionaries, sets) from existing ones. They are often more readable and faster than traditional `for` loops.

- **List Comprehension**: `[expression for item in iterable if condition]`
- Python

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

-
-

- **Dictionary Comprehension**: {key_expr: value_expr for item in iterable if condition}
- Python

squares_dict = {x: x*x for x in range(5)} # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

-
-
- **Set Comprehension**: {expression for item in iterable if condition}
- Python

even_numbers = {x for x in range(10) if x % 2 == 0} # {0, 2, 4, 6, 8}

-
-

34. What is with statement/context managers?

The with statement is used to simplify resource management, ensuring that a resource (like a file or a network connection) is properly closed or released, even if an error occurs. The with statement uses objects called **context managers**.

- A context manager has two special methods: __enter__ and __exit__.
- __enter__ is called when the with block is entered.
- __exit__ is called when the block is exited, regardless of whether it exited normally or due to an exception.

Example:

Python

```
with open("test.txt", "w") as file:
    file.write("Hello, World!")
# The file is automatically closed here, even if write() fails.
```

35. Difference between deep copy and shallow copy?

This is a duplicate of question 7.

- **Shallow Copy**: Creates a new object, but it doesn't create copies of the nested objects. It copies the references to the nested objects. If you modify a nested object in the copy, the original object will also be affected. Use copy.copy().
- **Deep Copy**: Creates a completely new object and recursively creates copies of all nested objects. The new object is fully independent of the original. If you modify the new object, the original remains unchanged. Use copy.deepcopy().

Example:

Python

```
import copy

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

# Shallow copy
shallow_copy = copy.copy(original_list)
shallow_copy[0][0] = 99 # Changes both lists
print(original_list) # [[99, 2], [3, 4]]

# Deep copy
deep_copy = copy.deepcopy(original_list)
deep_copy[1][0] = 88 # Only changes the deep copy
print(original_list) # [[99, 2], [3, 4]]
print(deep_copy)     # [[99, 2], [88, 4]]
```

36. What is monkey patching in Python?

Monkey patching is a technique where you dynamically modify or extend the behavior of a module, class, or function at runtime, without changing the original source code. This is usually done by adding, replacing, or modifying an existing method or attribute of a class or module. While it can be useful for testing or adding temporary fixes, it's often considered a bad practice as it can make code harder to understand and debug.

Example:

Python

```
# original_module.py
class MyClass:
    def greet(self):
        return "Hello!"

# patcher.py
from original_module import MyClass

def new_greet(self):
    return "Hi, I'm patched!"

MyClass.greet = new_greet # Monkey patching!

# main.py
from original_module import MyClass

obj = MyClass()
print(obj.greet()) # Hi, I'm patched!
```

37. What are metaclasses in Python?

A **metaclass** is a class of a class. It's what creates a class object. When you define a class in Python, the class itself is an object, and its type is `type`. `type` is the default metaclass in Python. You can create your own metaclasses to customize the creation process of classes, for example, to automatically add methods or attributes to all classes that use your metaclass. This is an advanced concept rarely needed in day-to-day programming.

Example:

```
Python

class MyMeta(type):
    def __new__(cls, name, bases, dct):
        dct['say_hi'] = lambda self: "Hi from a metaclass!"
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=MyMeta):
    pass

obj = MyClass()
print(obj.say_hi()) # Hi from a metaclass!
```

---

6. Python Memory & Performance

38. Explain Python's GIL (Global Interpreter Lock).

The **Global Interpreter Lock (GIL)** is a mutex (mutual exclusion lock) that protects access to Python objects, preventing multiple native threads from executing Python bytecodes at the same time. This means that even on a multi-core processor, only one thread can be executing Python code at any given time.

Impact:

- **CPU-bound tasks**: The GIL is a performance bottleneck for tasks that are CPU-intensive (e.g., heavy computations). A multi-threaded Python program won't use more than one CPU core for such tasks.
- **I/O-bound tasks**: For tasks that involve waiting for external resources (e.g., network requests, disk I/O), the GIL is not a bottleneck. When one thread is waiting for I/O, the GIL is released, allowing another thread to run.
- **Workarounds**: To achieve true parallelism for CPU-bound tasks, you use **multiprocessing** (which runs separate Python interpreters, each with its own GIL) instead of multithreading.

39. How does Python manage memory?

This is a repeat of question 8. See the answer for question 8 above.

40. What is memory leak in Python? How to avoid it?

A **memory leak** occurs when a program allocates memory but fails to release it when it's no longer needed, leading to a gradual consumption of available memory. In Python, memory leaks usually happen due to:

- **Circular references** that aren't garbage collected properly (though Python's GC is good at this).
- **Global variables** that hold references to large objects that are no longer needed.
- **Improper use of C extensions** that don't handle reference counts correctly.
- **Long-running processes** that accumulate data in memory.

How to avoid them:

- Use `del` to explicitly delete references to objects when they're no longer needed.
- Be mindful of circular references and break them if possible.
- Use weak references from the `weakref` module for objects that should not prevent garbage collection.
- Profile your application's memory usage to identify growing memory consumption.

41. How to optimize Python code performance?

- **Use built-in data structures and functions**: C-implemented built-ins like `list`, `dict`, `set`, and functions like `map`, `filter`, and `sum` are highly optimized.
- **Use comprehensions**: List, set, and dictionary comprehensions are often faster than `for` loops.
- **Choose the right data structure**: Dictionaries and sets offer O(1) average time complexity for lookups, which is much faster than lists (O(n)).
- **Use generators**: For large datasets, generators save memory by yielding one item at a time instead of building a full list.
- **Profile your code**: Use tools like `cProfile` and `memory_profiler` to identify bottlenecks.
- **Avoid excessive function calls**: Function calls have overhead.
- **Utilize `multiprocessing` for CPU-bound tasks** to bypass the GIL.
- **Use JIT compilers** like PyPy for computationally intensive code.

---

7. Multithreading & Multiprocessing

42. Difference between threading and multiprocessing in Python.

- **Threading**: Uses multiple threads within a single process. All threads share the same memory space. Due to the **GIL**, only one thread can execute Python bytecode at a time.
    - **Use case**: I/O-bound tasks (e.g., downloading files, network requests) where threads spend most of their time waiting.
- **Multiprocessing**: Creates separate processes, each with its own Python interpreter and memory space. This bypasses the GIL, allowing for true parallelism on multi-core CPUs. Communication between processes is done through inter-process communication (IPC).
    - **Use case**: CPU-bound tasks (e.g., complex calculations, data processing).

Analogy:

- **Threading** is like having a single chef (the process) with multiple hands (the threads) working on a dish. The chef can only do one thing at a time (due to GIL), but can switch between tasks quickly (e.g., chopping, stirring, seasoning).
- **Multiprocessing** is like having multiple chefs (the processes) in separate kitchens, each cooking their own dish. They can work in parallel on different tasks.

43. What is asyncio in Python?

`asyncio` is a library for writing concurrent code using the `async/await` syntax. It's a form of **cooperative multitasking** or **single-threaded concurrency**. A single thread can manage multiple I/O operations simultaneously by "awaiting" for an operation to complete, allowing another task to run in the meantime. `asyncio` is not about parallelism (using multiple CPU cores); it's about efficiency in handling many concurrent I/O operations.

Use Case: Ideal for I/O-bound tasks like web servers, network clients, and database access.

44. Difference between `concurrent.futures` and threading?

- **`concurrent.futures`** is a higher-level library that provides an easier way to use threads and processes. It simplifies concurrent programming by providing `ThreadPoolExecutor` and `ProcessPoolExecutor`.
- **`threading`** is a low-level library for creating and managing threads directly.
- **Key difference**: `concurrent.futures` offers a simpler, cleaner API (e.g., using `submit()` and `as_completed()`) for managing a pool of workers (threads or processes), while the `threading` module requires more manual management of threads. For most use cases, `concurrent.futures` is the preferred choice for its simplicity.

45. How does Python handle concurrency with GIL?

As discussed, the GIL prevents true parallelism for CPU-bound tasks in multithreading. However, it doesn't hinder concurrency for I/O-bound tasks.

- When a thread is waiting for an I/O operation (e.g., reading from a network socket), it **releases the GIL**, allowing another thread to acquire it and run.
- Once the I/O operation is complete, the original thread will try to re-acquire the GIL to continue its execution.
- This cooperative behavior allows multiple I/O-bound tasks to run "concurrently" even though only one thread is executing Python code at any given moment.

---

8. File Handling & I/O

46. How to read/write files in Python?

File I/O is done using the built-in `open()` function.

- **`open(filename, mode)`**: `filename` is the path to the file, and `mode` is a string indicating the purpose (`'r'` for read, `'w'` for write, `'a'` for append, `'x'` for exclusive

creation, 'b' for binary, 't' for text). The with statement is the best practice for file handling as it ensures the file is automatically closed.

- **Reading**:
- Python

```
with open("my_file.txt", "r") as f:
    content = f.read()  # reads the entire file
    lines = f.readlines() # reads all lines into a list
```

-
-
- **Writing**:
- Python

```
with open("new_file.txt", "w") as f:
    f.write("Hello, World!")
```

-
-
- **Appending**:
- Python

```
with open("new_file.txt", "a") as f:
    f.write("\nThis is a new line.")
```

-
-

47. Difference between binary mode and text mode in file handling?

- **Text Mode ('t')**: The default mode. When you read or write, Python performs **encoding/decoding**. It handles newline characters (\n) and other platform-specific line endings for you. It's used for text files (.txt, .csv, .py).
- **Binary Mode ('b')**: No encoding or decoding is performed. Data is read and written as raw bytes. This mode is essential for non-text files like images (.jpg), audio files (.mp3), or compressed files (.zip). You read and write using bytes objects.
- Python

```
with open("image.jpg", "rb") as f:
    data = f.read()
```

-
-

48. What is pickle in Python?

pickle is a module used for serializing and deserializing Python objects.

- **Serialization (pickling)**: The process of converting a Python object (e.g., a list, dictionary, or custom class instance) into a byte stream. This stream can be stored in a file or transmitted over a network.
- **Deserialization (unpickling)**: The reverse process of converting the byte stream back into a Python object.

Use cases: Caching objects, storing complex data in a file, or sending Python objects over a network.

- **Warning**: Pickled data can be unsafe if unpickled from an untrusted source, as it can execute arbitrary code.

Example:

```
Python

import pickle

data = {'name': 'Alice', 'age': 30}

# Pickling
with open('data.pkl', 'wb') as f:
    pickle.dump(data, f)

# Unpickling
with open('data.pkl', 'rb') as f:
    loaded_data = pickle.load(f)

print(loaded_data) # {'name': 'Alice', 'age': 30}
```

49. What is JSON handling in Python?

JSON (JavaScript Object Notation) is a lightweight data interchange format. Python's built-in `json` module provides a simple API for working with JSON data.

- **`json.dumps()`**: Serializes a Python object into a JSON formatted string.
- **`json.loads()`**: Deserializes a JSON formatted string into a Python object (usually a dictionary or list).
- **`json.dump()`**: Serializes and writes a Python object to a file.
- **`json.load()`**: Reads a JSON file and deserializes it into a Python object.

Example:

```
Python

import json

data = {'name': 'Alice', 'age': 30}
json_string = json.dumps(data)
```

```
print(json_string) # '{"name": "Alice", "age": 30}'

loaded_data = json.loads(json_string)
print(loaded_data['name']) # Alice
```

---

9. Python Libraries & Frameworks

50. What are popular Python libraries for data science (NumPy, Pandas, etc.)?

- **NumPy**: The fundamental library for numerical computing in Python. It provides powerful N-dimensional array objects (`ndarray`) and a vast collection of functions for high-performance mathematical operations.
- **Pandas**: A library for data manipulation and analysis. It provides two key data structures: `Series` (1D labeled array) and `DataFrame` (2D labeled table). Pandas simplifies data cleaning, transformation, and analysis.
- **Matplotlib**: A comprehensive library for creating static, animated, and interactive visualizations in Python.
- **Scikit-learn**: A machine learning library that provides simple and efficient tools for data mining and data analysis. It includes a wide range of algorithms for classification, regression, clustering, and more.
- **TensorFlow/PyTorch**: Open-source machine learning frameworks for building and training neural networks.

51. What is Django vs Flask?

- **Django**: A high-level, "batteries-included" web framework. It comes with everything you need out of the box, including an ORM, an admin panel, a template engine, and a routing system. It follows the **Model-View-Template (MVT)** pattern and is great for building large, complex, and secure web applications quickly.
- **Flask**: A minimalist, micro-framework. It provides the bare essentials for web development, such as routing and request handling. It's up to the developer to choose and integrate additional libraries for things like databases, authentication, or forms. Flask is ideal for small to medium-sized projects, microservices, and APIs where you need more control and flexibility.

52. What is FastAPI?

FastAPI is a modern, high-performance web framework for building APIs with Python. It's built on top of standard Python type hints and is designed for fast development and production-ready performance.

- **Key features**:
    - **High performance**: On par with Node.js and Go.
    - **Automatic data validation and serialization**: Uses Pydantic to ensure data is correct.

    - **Automatic interactive API documentation**: Generates OpenAPI (Swagger) and ReDoc documentation automatically.
    - **Asynchronous support**: Built with `async/await` for high-concurrency I/O operations.

53. What are ORM frameworks in Python?

An **ORM (Object-Relational Mapper)** is a library that allows you to interact with a database using Python objects instead of writing raw SQL queries. It provides an abstraction layer between your Python code and the database.

- **How it works**:
    - You define Python classes that represent your database tables.
    - The ORM handles the translation of object-oriented operations (e.g., creating an object, saving it) into SQL queries behind the scenes.
- **Benefits**:
    - Reduces the need to write repetitive SQL.
    - Code is more readable and maintainable.
    - Helps prevent SQL injection attacks.
- **Examples**: **SQLAlchemy** (very popular and powerful) and Django's built-in ORM.

---

10. Testing in Python

54. What is unittest in Python?

`unittest` is Python's built-in unit testing framework, inspired by JUnit. It provides a simple, object-oriented way to organize and run tests.

- **Key components**:
    - **Test Fixture**: A setup/teardown function (e.g., `setUp`, `tearDown`) that prepares and cleans up resources for a test.
    - **Test Case**: A single test class that inherits from `unittest.TestCase`. Each method starting with `test_` is a test.
    - **Test Suite**: A collection of test cases.
    - **Test Runner**: An object that executes tests and reports the results.
- **Example**:
- Python

```
import unittest

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

class TestAdd(unittest.TestCase):
    def test_add_positive(self):
        self.assertEqual(add(2, 3), 5)
```

```
    def test_add_negative(self):
        self.assertEqual(add(-1, -1), -2)
```

55. Difference between unittest and pytest?

- **unittest**:
  - **"Batteries-included"**: Part of the Python standard library.
  - **Verbose**: Requires you to write boilerplate code (setUp, TestCase classes, etc.).
  - **Assertions**: Uses self.assertEqual(), self.assertTrue(), etc.
- **pytest**:
  - **Simpler**: Uses a more straightforward, less verbose syntax. You can write simple functions for tests (def test_...) without needing a class.
  - **Assertions**: Uses standard assert statements (assert x == y).
  - **Plugins**: Has a rich ecosystem of plugins for things like code coverage, mocking, and parallel test execution.
  - **pytest is generally the preferred choice for new projects** due to its simplicity and powerful features.

56. How to mock objects in Python tests?

Mocking is a technique used in unit testing to replace complex or slow dependencies (like a database, an external API, or a slow function) with "mock" objects. This allows you to test your code in isolation without relying on external systems.

- Python's built-in unittest.mock module provides the Mock and patch classes for this purpose.
- **unittest.mock.patch**: A decorator or context manager used to temporarily replace an object with a mock object.
- **Example**:
- Python

```
from unittest.mock import patch

def get_weather():
    # This function calls a slow external API
    pass

@patch('my_module.get_weather')
def test_my_function_with_mock(mock_get_weather):
    # Configure the mock to return a fake value
    mock_get_weather.return_value = 'Sunny'
    # Now call the function that uses get_weather
    result = my_function()
    # The mock was used, not the real function
```

```
assert result == 'It is Sunny'
```

---

11. Python in Databases & APIs

57. How to connect Python with MySQL/PostgreSQL?

You typically use a database connector library.

1. **Install the library**:
   - For MySQL: `pip install mysql-connector-python` or `pip install pymysql`.
   - For PostgreSQL: `pip install psycopg2-binary`.
2. **Import and connect**: Use the library to establish a connection.
3. **Create a cursor**: A cursor is used to execute SQL queries.
4. **Execute queries**: Use the cursor to execute `SELECT`, `INSERT`, `UPDATE`, etc.
5. **Fetch results**: Use methods like `fetchone()` or `fetchall()` to retrieve data.
6. **Commit changes**: For `INSERT`/`UPDATE`, you must `commit()` the changes.
7. **Close connection**: Close the cursor and the connection to free up resources.

Example (PostgreSQL):

```python
import psycopg2

conn = psycopg2.connect("dbname=test user=admin password=secret")
cur = conn.cursor()
cur.execute("SELECT * FROM users;")
rows = cur.fetchall()
for row in rows:
    print(row)
cur.close()
conn.close()
```

58. What is SQLAlchemy?

SQLAlchemy is a powerful and flexible ORM (Object-Relational Mapper) library for Python. It provides a full suite of well-known persistence patterns and is known for its high performance and adaptability. It has two main components:

- **Core**: A low-level toolkit for database access that provides a SQL Expression Language. It's a great way to build and execute SQL queries programmatically.
- **ORM**: A high-level component that maps Python classes to database tables and allows you to interact with the database using Python objects.

SQLAlchemy is database-agnostic, meaning it works with a wide variety of databases like MySQL, PostgreSQL, SQLite, and Oracle.

59. How to consume REST APIs in Python?

To consume a REST API, you need to make HTTP requests (GET, POST, PUT, DELETE, etc.). The **requests** library is the de facto standard for this in Python.

1. **Install requests**: pip install requests.
2. **Import**: import requests.
3. **Make a request**: Use requests.get(), requests.post(), etc., to make a request to the API endpoint.
4. **Handle the response**: The response object contains the status code, headers, and the body of the response. Use response.json() to parse the JSON content into a Python dictionary.

Example (GET request):

```
Python

import requests

response = requests.get("https://jsonplaceholder.typicode.com/posts/1")
if response.status_code == 200:
    data = response.json()
    print(data['title'])
else:
    print("Failed to fetch data.")
```

---

12. Python Interview Coding Questions

60. Reverse a string without using slicing.

```
Python

def reverse_string(s):
    reversed_str = ""
    for char in s:
        reversed_str = char + reversed_str
    return reversed_str

print(reverse_string("hello")) # olleh
```

61. Find the first non-repeating character in a string.

```
Python

from collections import Counter

def first_non_repeating_char(s):
    count = Counter(s)
```

```
    for char in s:
        if count[char] == 1:
            return char
    return None

print(first_non_repeating_char("swiss")) # w
```

62. Check if a string is a palindrome.

```
Python

def is_palindrome(s):
    return s == s[::-1]

print(is_palindrome("madam")) # True
print(is_palindrome("hello")) # False
```

63. Find factorial using recursion.

```
Python

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

print(factorial(5)) # 120 (5 * 4 * 3 * 2 * 1)
```

64. Implement Fibonacci sequence.

```
Python

def fibonacci(n):
    a, b = 0, 1
    result = []
    while a < n:
        result.append(a)
        a, b = b, a + b
    return result

print(fibonacci(10)) # [0, 1, 1, 2, 3, 5, 8]
```

65. Remove duplicates from a list.

```
Python

def remove_duplicates(my_list):
    return list(set(my_list))
```

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

66. Find the second largest number in a list.

```
Python

def second_largest(numbers):
    unique_numbers = sorted(list(set(numbers)), reverse=True)
    if len(unique_numbers) >= 2:
        return unique_numbers[1]
    return None

print(second_largest([1, 5, 2, 8, 8])) # 5
```

67. Count frequency of words in a string.

```
Python

from collections import Counter

def word_frequency(text):
    words = text.lower().split()
    return Counter(words)

text = "Hello world, hello Python, hello world."
print(word_frequency(text))
# Counter({'hello': 3, 'world,': 2, 'python,': 1})
```

68. Implement binary search in Python.

```
Python

def binary_search(sorted_list, target):
    low = 0
    high = len(sorted_list) - 1
    while low <= high:
        mid = (low + high) // 2
        guess = sorted_list[mid]
        if guess == target:
            return mid
        if guess > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1 # Target not found

my_list = [1, 3, 5, 7, 9, 11]
```

```
print(binary_search(my_list, 7))  # 3
print(binary_search(my_list, 4))  # -1
```

69. Check if two strings are anagrams.

```
Python

def are_anagrams(s1, s2):
    return sorted(s1.lower()) == sorted(s2.lower())

print(are_anagrams("listen", "silent")) # True
print(are_anagrams("hello", "world"))  # False
```

70. Flatten a nested list in Python.

```
Python

def flatten_list(nested_list):
    flat_list = []
    for item in nested_list:
        if isinstance(item, list):
            flat_list.extend(flatten_list(item))
        else:
            flat_list.append(item)
    return flat_list

nested = [1, [2, 3], [4, [5, 6]]]
print(flatten_list(nested)) # [1, 2, 3, 4, 5, 6]
```