Python

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Explain the Global Interpreter Lock (GIL) and its implications for multi-threaded Python applications.

The Global Interpreter Lock (GIL) is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecode simultaneously. This means that in CPython, even on multi-core systems, only one thread executes Python code at a time.

Key Implications:

- **CPU-bound tasks:** Multi-threading provides no performance benefit; threads take turns executing
- **I/O-bound tasks:** Threading still helps because threads release the GIL during I/O operations
- **Workarounds:** Use multiprocessing for CPU-bound parallelism, or async/await for I/O concurrency
- **Alternative implementations:** Jython and IronPython don't have a GIL

Example showing GIL impact:

```
import threading
import time

def cpu_task():
    count = 0
    for i in range(10**7):
        count += 1

# Threading doesn't speed up CPU-bound work
threads = [threading.Thread(target=cpu_task) for _ in range(4)]
start = time.time()
for t in threads: t.start()
for t in threads: t.join()
print(f"Time: {time.time()-start:.2f}s")
```

Best practice: Use threading for I/O operations, multiprocessing for CPU-intensive tasks, and asyncio for high-concurrency I/O.

2. How do Python descriptors work, and what are their practical use cases?

Descriptors are objects that define how attribute access is handled through the **__get__**, **__set__**, and **__delete__** methods. They implement the descriptor protocol and are fundamental to Python's property system, methods, and class/static methods.

Descriptor Protocol:

- **__get__(self, obj, type):** Called when attribute is accessed
- **__set__(self, obj, value):** Called when attribute is assigned
- **__delete__(self, obj):** Called when attribute is deleted

Practical Example - Type Validation:

```
class TypedProperty:
    def __init__(self, expected_type):
        self.expected_type = expected_type
        self.data = {}
    def __get__(self, obj, type):
        return self.data.get(obj)

    def __set__(self, obj, value):
        if not isinstance(value, self.expected_type):
            raise TypeError(f"Expected {self.expected_type}")
        self.data[obj] = value
```

Common Use Cases:

- **Validation:** Enforce type checking or value constraints
- **Lazy loading:** Compute values only when accessed
- **ORM frameworks:** Django and SQLAlchemy use descriptors for model fields
- **Property decorators:** @property is implemented using descriptors

3. What is the difference between __new__ and __init__, and when would you override __new__?

__new__ is a static method responsible for creating and returning a new instance, while **__init__** initializes an already-created instance. __new__ runs before __init__ and is rarely overridden except in specific scenarios.

Key Differences:

- **__new__(cls, ...):** Creates the instance, must return an object
- **__init__(self, ...):** Initializes the instance, returns None
- __new__ receives the class as first argument; __init__ receives the instance

When to Override __new__:

- **Singleton pattern:** Control instance creation to ensure only one exists
- **Immutable types:** Subclassing str, int, tuple requires __new__
- **Metaclass-like behavior:** Modify instance before __init__ runs
- **Factory patterns:** Return different types based on arguments

Singleton Example:

```
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

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

s1 = Singleton(10)
s2 = Singleton(20)
print(s1 is s2)  # True
```

4. Explain Python's memory management and garbage collection mechanisms.

Python uses automatic memory management with reference counting as the primary mechanism and a generational garbage collector to handle circular references.

Reference Counting:

- Each object maintains a count of references pointing to it
- When count reaches zero, memory is immediately deallocated
- Fast and deterministic, but can't handle circular references
- Check with **sys.getrefcount(obj)**

Generational Garbage Collection:

- **Generation 0:** Newly created objects, collected most frequently
- **Generation 1:** Objects that survived one GC cycle
- **Generation 2:** Long-lived objects, collected least frequently
- Detects and breaks circular references

Example of Circular Reference:

```
import gc

class Node:
    def __init__(self):
        self.ref = None

a = Node()
b = Node()
a.ref = b
b.ref = a  # Circular reference

del a, b  # Reference count > 0, needs GC
gc.collect()  # Manually trigger collection
```

Memory Optimization Tips:

- Use **__slots__** to reduce per-instance memory overhead
- Implement **__del__** carefully to avoid resurrection issues
- Use weak references (weakref module) to avoid circular refs
- Profile with tracemalloc or memory_profiler

5. How do metaclasses work in Python, and what problems do they solve?

Metaclasses are classes of classes—they define how classes behave. When you create a class, Python uses a metaclass (by default **type**) to construct it. Metaclasses allow you to intercept class creation and modify class behavior.

How They Work:

- Classes are instances of metaclasses
- Default metaclass is **type**
- Define custom metaclass by inheriting from type
- Override **__new__** or **__init__** to customize class creation

Example - Enforcing Method Implementation:

```
class InterfaceMeta(type):
    def __new__(cls, name, bases, dct):
        if name != 'Interface':
            if 'required_method' not in dct:
                raise TypeError(f"{name} must implement required_method")
        return super().__new__(cls, name, bases, dct)

class Interface(metaclass=InterfaceMeta):
    pass

class Valid(Interface):
    def required_method(self): pass
```

Common Use Cases:

- **ORMs:** Django models use metaclasses to register fields
- **Singleton enforcement:** Control instance creation at class level
- **API validation:** Ensure classes implement required interfaces
- **Automatic registration:** Register classes in a registry upon creation
- **Attribute modification:** Automatically add or modify class attributes

Note: Metaclasses are powerful but complex. Consider alternatives like class decorators or __init_subclass__ for simpler cases.

6. Explain the differences between @staticmethod, @classmethod, and instance methods.

These three method types differ in what they receive as their first argument and how they interact with class and instance state.

Instance Methods:

- Receive **self** (the instance) as first parameter
- Can access and modify instance state
- Can access class state through self.__class__
- Most common method type

Class Methods (@classmethod):

- Receive **cls** (the class) as first parameter
- Can access and modify class state
- Cannot access instance-specific data
- Useful for factory methods and alternative constructors

Static Methods (@staticmethod):

- Don't receive implicit first parameter
- Cannot access instance or class state directly
- Behave like regular functions but belong to class namespace
- Used for utility functions related to the class

Practical Example:

```
class Date:
    def __init__(self, year, month, day):
        self.year, self.month, self.day = year, month, day

    @classmethod
    def from_string(cls, date_str):
        y, m, d = map(int, date_str.split('-'))
        return cls(y, m, d)  # Factory method

    @staticmethod
    def is_valid(year, month, day):
        return 1 <= month <= 12 and 1 <= day <= 31
```

7. What are context managers, and how do you implement custom ones?

Context managers manage resources by defining setup and teardown actions using the **with** statement. They ensure cleanup code runs even if exceptions occur, following the Resource Acquisition Is Initialization (RAII) pattern.

Protocol Methods:

- **__enter__(self):** Called when entering the with block, returns resource
- **__exit__(self, exc_type, exc_val, exc_tb):** Called when exiting, handles cleanup
- __exit__ receives exception info if one occurred
- Return True from __exit__ to suppress exceptions

Class-Based Implementation:

```
class DatabaseConnection:
    def __init__(self, conn_str):
        self.conn_str = conn_str

    def __enter__(self):
        self.conn = create_connection(self.conn_str)
        return self.conn

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.conn.close()
        return False  # Don't suppress exceptions
```

Using contextlib Decorator:

```
from contextlib import contextmanager

@contextmanager
def timer():
    start = time.time()
    yield
    print(f"Elapsed: {time.time()-start:.2f}s")

with timer():
    expensive_operation()
```

Common Use Cases:

- File handling, database connections, locks
- Transaction management
- Timing and profiling code blocks
- Temporary state changes

8. How does Python's async/await work, and what is the event loop?

Async/await enables cooperative multitasking for I/O-bound operations. Instead of blocking, async functions (coroutines) yield control to an event loop, which manages multiple operations concurrently on a single thread.

Key Concepts:

- **async def:** Defines a coroutine function that returns a coroutine object
- **await:** Suspends coroutine execution until awaited operation completes
- **Event loop:** Scheduler that runs coroutines and handles I/O operations
- **Concurrency not parallelism:** Single-threaded but handles multiple operations

Basic Example:

```
import asyncio

async def fetch_data(url, delay):
    await asyncio.sleep(delay)  # Non-blocking
    return f"Data from {url}"

async def main():
    results = await asyncio.gather(
        fetch_data("api1", 2),
        fetch_data("api2", 1)
    )
    print(results)

asyncio.run(main())
```

When to Use Async:

- **I/O-bound tasks:** Network requests, file operations, database queries
- **High concurrency:** Thousands of simultaneous connections
- **Not for CPU-bound:** Use multiprocessing instead

Event Loop Mechanics:

- Maintains queue of ready coroutines
- Executes coroutines until they await
- Monitors I/O operations using selectors
- Resumes coroutines when I/O completes

9. Explain Python's descriptor protocol and how properties are implemented using it.

The **descriptor protocol** allows objects to customize attribute access. The **@property** decorator is actually implemented using descriptors, providing a Pythonic way to create managed attributes with getter, setter, and deleter methods.

Property Implementation:

- property() is a descriptor class
- Stores getter, setter, and deleter functions
- __get__ calls the getter function
- __set__ calls the setter function

Using @property Decorator:

```
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
```

Equivalent Descriptor Implementation:

```
class CelsiusDescriptor:
    def __init__(self):
        self.data = {}

    def __get__(self, obj, type):
        return self.data.get(obj, 0)

    def __set__(self, obj, value):
        if value < -273.15:
            raise ValueError("Below absolute zero")
        self.data[obj] = value
```

Benefits of Properties:

- Encapsulation without changing interface
- Computed attributes
- Validation and type checking
- Backward compatibility when refactoring

10. What are Python's data model special methods (dunder methods), and how do they enable operator overloading?

Special methods (magic methods or dunder methods) are surrounded by double underscores and define how objects behave with built-in operations. They enable operator overloading, making custom objects behave like built-in types.

Common Categories:

- **Initialization:** __init__, __new__, __del__
- **Representation:** __str__, __repr__, __format__
- **Comparison:** __eq__, __lt__, __gt__, __le__, __ge__, __ne__
- **Arithmetic:** __add__, __sub__, __mul__, __truediv__, __mod__
- **Container:** __len__, __getitem__, __setitem__, __contains__
- **Callable:** __call__
- **Context:** __enter__, __exit__

Example - Vector Class:

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

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

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

Best Practices:

- Always implement __repr__ for debugging
- Use @functools.total_ordering for comparison methods
- Return NotImplemented for unsupported operations
- Keep behavior consistent with built-in types

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement an LRU (Least Recently Used) Cache with O(1) time complexity for both get and put operations.

Solution: Use a combination of a **doubly linked list** and a **hash map**. The hash map stores key-node pairs for O(1) lookup, while the doubly linked list maintains access order.

```
class Node:
    def __init__(self, key, val):
        self.key, self.val = key, val
        self.prev = self.next = None

class LRUCache:
    def __init__(self, capacity):
        self.cap = capacity
        self.cache = {}
        self.left, self.right = Node(0, 0), Node(0, 0)
        self.left.next, self.right.prev = self.right, self.left
```

- **get(key):** Return value and move node to most recent position
- **put(key, value):** Insert/update and evict LRU if at capacity
- **Time Complexity:** O(1) for both operations
- **Space Complexity:** O(capacity)

2. What is the time complexity of dictionary operations in Python, and how does it handle collisions?

Dictionary Time Complexities:

- **Average case:** O(1) for get, set, delete operations
- **Worst case:** O(n) when hash collisions occur
- **Space complexity:** O(n)

Collision Handling

Python uses **open addressing with random probing** (not chaining). When a collision occurs:

- Uses a pseudo-random probe sequence to find the next available slot
- Hash table resizes when 2/3 full to maintain performance
- Deleted entries are marked as dummy to preserve probe sequences

Key requirement: Dictionary keys must be hashable (immutable types like int, str, tuple).

3. Find all pairs in an array that sum to a target value. What's the optimal approach?

Optimal Solution: Use a **hash set** for O(n) time complexity.

```
def find_pairs(arr, target):
    seen = set()
    pairs = []
    for num in arr:
        complement = target - num
        if complement in seen:
            pairs.append((complement, num))
        seen.add(num)
    return pairs
```

- **Time Complexity:** O(n) - single pass through array

- **Space Complexity:** O(n) - for the hash set
- **Alternative:** Two-pointer approach on sorted array O(n log n) but O(1) extra space
- Handles duplicates and negative numbers correctly

4. Implement a Min Stack that supports push, pop, top, and getMin operations all in O(1) time.

Solution: Maintain two stacks - one for values and one for tracking minimums.

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, val):
        self.stack.append(val)
        min_val = min(val, self.min_stack[-1] if self.min_stack else val)
        self.min_stack.append(min_val)

    def getMin(self):
        return self.min_stack[-1]
```

- **Time Complexity:** O(1) for all operations
- **Space Complexity:** O(n) for storing two stacks
- **Key insight:** min_stack stores the minimum value at each level
- **Alternative:** Store tuples (value, current_min) in single stack

5. Explain the difference between sets and frozensets in Python. When would you use each?

Sets vs Frozensets

Set (mutable):

- Can add/remove elements after creation
- Not hashable - cannot be dict keys or set elements
- Methods: add(), remove(), discard(), pop()

Frozenset (immutable):

- Cannot be modified after creation
- Hashable - can be dict keys or elements of other sets
- Only supports query operations

```
regular_set = {1, 2, 3}
frozen = frozenset([1, 2, 3])
set_of_sets = {frozenset([1, 2]), frozenset([3, 4])}
dict_key = {frozen: 'value'}
```

Use frozenset when: Need set as dictionary key, set element, or guarantee immutability.

6. Implement a sliding window maximum algorithm to find the maximum in every window of size k.

Optimal Solution: Use a **deque (double-ended queue)** to maintain indices of potential maximums.

```
from collections import deque

def max_sliding_window(nums, k):
    result = []
    dq = deque()
    for i, num in enumerate(nums):
        while dq and nums[dq[-1]] < num:
            dq.pop()
        dq.append(i)
        if dq[0] <= i - k:
            dq.popleft()
        if i >= k - 1:
```

```
        result.append(nums[dq[0]])
    return result
```

- **Time Complexity:** O(n) - each element added/removed once
- **Space Complexity:** O(k) - deque stores at most k elements
- Deque maintains decreasing order of values

7. What is the internal implementation of Python lists, and how does append() achieve amortized O(1) complexity?

Python List Implementation

Internal structure:

- Dynamic array (resizable array) implemented in C
- Stores pointers to Python objects, not objects themselves
- Maintains allocated size separate from logical size

Amortized O(1) Append

Strategy: Over-allocate memory when resizing

- When capacity reached, allocates ~12.5% more space
- New capacity formula: new_size = (old_size >> 3) + 6
- Most appends are O(1); occasional O(n) resize
- Amortized analysis: total cost divided by operations = O(1)

Other complexities:

- insert(0, x): O(n) - shifts all elements
- pop(): O(1), pop(0): O(n)

8. Implement a function to detect a cycle in a linked list using O(1) space complexity.

Solution: Use **Floyd's Cycle Detection Algorithm** (tortoise and hare).

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False
```

- **Time Complexity:** O(n) - visits each node at most once
- **Space Complexity:** O(1) - only two pointers
- **How it works:** If cycle exists, fast pointer will eventually meet slow
- **Extension:** To find cycle start, reset one pointer to head after meeting

9. Compare the performance characteristics of list, deque, and array.array in Python.

Performance Comparison

list (dynamic array):

- Append/pop from end: O(1) amortized
- Insert/delete at beginning: O(n)
- Random access: O(1)
- Stores object references - flexible but memory overhead

collections.deque (doubly-linked list):

- Append/pop from both ends: O(1)

- Random access: O(n) - slow indexing
- Best for queue/stack operations

array.array (typed array):

- Similar to list but stores primitive types compactly
- Memory efficient for numeric data
- Same time complexities as list

```
from collections import deque
from array import array

d = deque([1, 2, 3])
arr = array('i', [1, 2, 3])
```

10. Implement a Trie (prefix tree) for efficient string search operations. What are the time complexities?

Trie Implementation:

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True
```

Time Complexities

- **Insert:** O(m) where m is word length
- **Search:** O(m) for exact match
- **StartsWith (prefix):** O(m)
- **Space:** O(ALPHABET_SIZE * N * M) worst case

Use cases: Autocomplete, spell checker, IP routing tables.

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly. What are the key components and how would you handle billions of URLs?

Core Requirements

- **Functional:** Generate short URLs, redirect to original URLs, optional custom aliases
- **Non-functional:** High availability, low latency, scalability to billions of URLs

Architecture Components

- **API Gateway:** Rate limiting, authentication, request routing
- **Application Servers:** Stateless services for URL generation and lookup
- **Database:** NoSQL (Cassandra/DynamoDB) for horizontal scaling
- **Cache Layer:** Redis/Memcached for hot URLs (80/20 rule)
- **CDN:** Geographic distribution for faster redirects

URL Generation Strategy

```
import hashlib
import base62

def generate_short_url(long_url, counter):
    hash_input = long_url + str(counter)
    hash_val = hashlib.md5(hash_input.encode()).hexdigest()
    return base62.encode(int(hash_val[:8], 16))[:7]
```

Database Schema

- **Partition key:** short_url (for fast lookups)
- **Attributes:** original_url, created_at, expiry, user_id, click_count

Scalability Considerations

- **Write scaling:** Use distributed ID generation (Snowflake/UUID)
- **Read scaling:** Cache popular URLs, use read replicas
- **Storage:** ~7 chars base62 = 62^7 = 3.5 trillion combinations
- **CAP theorem:** Choose AP (availability + partition tolerance) over consistency

2. How would you design a real-time chat application supporting millions of concurrent users? Discuss WebSocket management, message delivery guarantees, and scaling strategies.

Architecture Overview

- **WebSocket Gateway:** Manages persistent connections (Socket.io/WebSocket)
- **Message Queue:** Kafka/RabbitMQ for reliable message delivery
- **Presence Service:** Redis for online/offline status
- **Message Storage:** Cassandra for chat history (time-series data)
- **Media Storage:** S3/CDN for files and images

Connection Management

```
class ConnectionManager:
    def __init__(self):
        self.active_connections = {}
    async def connect(self, user_id, websocket):
        self.active_connections[user_id] = websocket
        await self.broadcast_presence(user_id, 'online')

    async def send_message(self, user_id, message):
        ws = self.active_connections.get(user_id)
        if ws:
            await ws.send_json(message)
```

Message Delivery Guarantees

- **At-least-once:** ACK mechanism with retry logic
- **Message IDs:** Unique IDs for deduplication
- **Offline queue:** Store messages in DB when user offline
- **Read receipts:** Separate acknowledgment layer

Scaling Strategies

- **Horizontal scaling:** Multiple WebSocket servers behind load balancer
- **Sticky sessions:** Route user to same server (consistent hashing)
- **Service mesh:** Server-to-server communication for cross-server messages
- **Sharding:** Partition users across servers by user_id
- **Rate limiting:** Prevent spam and abuse

Data Consistency

- Use event sourcing for message ordering
- Vector clocks for conflict resolution
- Eventually consistent reads acceptable for chat history

3. Design a distributed task queue system like Celery. How would you handle task scheduling, retries, priority queues, and worker management?

Core Components

- **Message Broker:** RabbitMQ/Redis for task queue
- **Result Backend:** Redis/PostgreSQL for task results
- **Workers:** Process pool executing tasks
- **Scheduler:** Periodic task execution (cron-like)
- **Monitoring:** Flower/Prometheus for observability

Task Queue Implementation

```
import json
import redis

class TaskQueue:
    def __init__(self, redis_client):
        self.redis = redis_client

    def enqueue(self, task_name, args, priority=0):
        task = {'name': task_name, 'args': args}
        queue = f'queue:priority:{priority}'
        self.redis.lpush(queue, json.dumps(task))

    def dequeue(self):
        for p in range(10, -1, -1):
            task = self.redis.brpop(f'queue:priority:{p}')
            if task: return json.loads(task[1])
```

Retry Mechanism

- **Exponential backoff:** delay = base_delay * (2 ^ retry_count)
- **Max retries:** Configurable limit per task
- **Dead letter queue:** Failed tasks after max retries
- **Idempotency:** Tasks should be safe to retry

Worker Management

- **Auto-scaling:** Spawn workers based on queue depth
- **Graceful shutdown:** Finish current task before stopping
- **Health checks:** Heartbeat mechanism to detect dead workers
- **Prefetch limit:** Control concurrent tasks per worker

Priority Queues

- Separate queues per priority level (0-10)
- Workers poll high-priority queues first
- Prevent starvation with weighted round-robin

Scheduling

- **Cron-like:** Store schedule in DB, beat process enqueues tasks
- **Delay queues:** Use sorted sets with timestamp as score
- **Time zones:** Store UTC, convert for execution

4. Design a news feed system like Twitter or Facebook. How would you generate, rank, and deliver personalized feeds to millions of users efficiently?

Feed Generation Approaches

- **Fanout-on-write (push):** Pre-compute feeds when post created
- **Fanout-on-read (pull):** Compute feed when user requests
- **Hybrid:** Push for active users, pull for inactive/celebrities

Architecture Components

- **Post Service:** Create and store posts
- **Fanout Service:** Distribute posts to followers' feeds
- **Feed Service:** Retrieve and rank feed items
- **Graph Service:** Manage follow relationships
- **Cache:** Redis for hot feeds (recent + popular)
- **Storage:** Cassandra for feed data (timeline)

Feed Generation (Fanout-on-write)

```
def fanout_post(post_id, author_id):
    followers = graph_service.get_followers(author_id)
    for follower_id in followers:
        feed_key = f'feed:{follower_id}'
        cache.zadd(feed_key, {post_id: timestamp})
        cache.zremrangebyrank(feed_key, 0, -1001)
    if len(followers) > 10000:
        mark_for_pull_model(author_id)
```

Ranking Algorithm

- **Signals:** Recency, engagement (likes, comments), author relevance, content type
- **ML model:** Predict engagement probability
- **Score formula:** weighted_sum(signals) with decay function
- **Personalization:** User interests, interaction history

Scalability Optimizations

- **Celebrity problem:** Don't fanout for users with millions of followers, use pull
- **Pagination:** Cursor-based for consistent results
- **Pre-aggregation:** Compute top posts periodically
- **Edge caching:** CDN for static content

Data Model

- **Feed table:** (user_id, post_id, timestamp) - partition by user_id
- **TTL:** Expire old feed entries automatically
- **Denormalization:** Store post metadata in feed for fast reads

5. How would you design a rate limiter service that can be used across multiple microservices? Discuss algorithms, distributed considerations, and implementation.

Rate Limiting Algorithms

- **Token Bucket:** Refill tokens at fixed rate, consume on request
- **Leaky Bucket:** Process requests at constant rate
- **Fixed Window:** Count requests per time window
- **Sliding Window Log:** Track timestamps of requests
- **Sliding Window Counter:** Hybrid approach (most accurate + efficient)

Token Bucket Implementation

```
import time

class TokenBucket:
    def __init__(self, capacity, refill_rate):
        self.capacity = capacity
        self.tokens = capacity
        self.refill_rate = refill_rate
        self.last_refill = time.time()

    def allow_request(self):
        self._refill()
        if self.tokens >= 1:
            self.tokens -= 1
            return True
        return False

    def _refill(self):
        now = time.time()
        tokens_to_add = (now - self.last_refill) * self.refill_rate
        self.tokens = min(self.capacity, self.tokens + tokens_to_add)
        self.last_refill = now
```

Distributed Rate Limiter

- **Centralized store:** Redis with atomic operations (INCR, EXPIRE)
- **Lua scripts:** Ensure atomicity for complex operations
- **Key structure:** rate_limit:{user_id}:{endpoint}:{window}

Redis Implementation

```
def is_allowed(user_id, limit, window):
    key = f'rate:{user_id}:{window}'
    pipe = redis.pipeline()
    pipe.incr(key)
    pipe.expire(key, window)
    result = pipe.execute()
    return result[0] <= limit
```

Design Considerations

- **Granularity:** Per-user, per-IP, per-API-key, per-endpoint
- **Rate limit tiers:** Different limits for free/premium users
- **Burst handling:** Allow temporary spikes
- **Distributed consensus:** Accept slight inaccuracy for performance

Response Headers

- X-RateLimit-Limit: Maximum requests allowed
- X-RateLimit-Remaining: Requests left in window
- X-RateLimit-Reset: Time when limit resets
- Retry-After: Seconds to wait when rate limited

6. Design a distributed caching system. How would you handle cache invalidation, consistency, eviction policies, and hot key problems?

Architecture Layers

- **L1 Cache:** In-memory application cache (local)
- **L2 Cache:** Distributed cache (Redis/Memcached)
- **L3 Cache:** CDN for static assets

Cache Strategies

- **Cache-aside:** App checks cache, loads from DB on miss
- **Read-through:** Cache loads from DB automatically
- **Write-through:** Write to cache and DB synchronously
- **Write-behind:** Write to cache, async write to DB

Cache-Aside Implementation

```
def get_user(user_id):
    cache_key = f'user:{user_id}'
    user = cache.get(cache_key)
    if user is None:
        user = db.query('SELECT * FROM users WHERE id=%s', user_id)
        if user:
            cache.set(cache_key, user, ttl=3600)
    return user

def update_user(user_id, data):
    db.update('users', user_id, data)
    cache.delete(f'user:{user_id}')
```

Cache Invalidation Strategies

- **TTL-based:** Set expiration time on all keys
- **Event-based:** Invalidate on data updates (pub/sub)
- **Version-based:** Include version in cache key
- **Tag-based:** Group related keys, invalidate by tag

Eviction Policies

- **LRU (Least Recently Used):** Remove oldest accessed items
- **LFU (Least Frequently Used):** Remove least accessed items
- **FIFO:** Remove oldest inserted items
- **Random:** Remove random items

Hot Key Problem

- **Local caching:** Cache hot keys in application memory
- **Replication:** Replicate hot keys across multiple cache nodes
- **Key splitting:** Split hot key into multiple keys with random suffix
- **Circuit breaker:** Fallback to DB when cache overloaded

Consistency Considerations

- **Eventual consistency:** Accept stale data for performance
- **Cache stampede:** Use locks to prevent thundering herd
- **Probabilistic early expiration:** Refresh before TTL expires

7. Design a search autocomplete system like Google's search suggestions. How would you handle prefix matching, ranking, and real-time updates at scale?

Data Structures

- **Trie (Prefix Tree):** Efficient prefix matching
- **Inverted Index:** Map prefixes to suggestions
- **Sorted Set:** Redis ZSET for ranking by popularity

Trie Implementation

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False
        self.frequency = 0
        self.suggestions = []

class Autocomplete:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word, freq):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True
        node.frequency = freq

    def search(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return []
            node = node.children[char]
        return self._get_suggestions(node, prefix)
```

Architecture Components

- **Query Service:** Handle autocomplete requests
- **Aggregation Service:** Collect and process query logs
- **Trie Builder:** Build and update trie periodically
- **Cache Layer:** Redis for popular prefixes
- **Storage:** Store trie snapshots and query logs

Ranking Signals

- **Query frequency:** Historical search volume
- **Recency:** Recent trending queries weighted higher
- **Personalization:** User's search history
- **Location:** Geographic relevance
- **Time:** Seasonal/time-based trends

Scalability Optimizations

- **Prefix caching:** Cache top suggestions for common prefixes
- **Sharding:** Distribute trie by first character or hash
- **Compression:** Use compressed trie (PATRICIA tree)
- **Lazy loading:** Load trie branches on demand
- **CDN:** Serve static suggestions from edge

Real-time Updates

- **Batch processing:** Update trie every few hours with Spark
- **Incremental updates:** Stream processing with Kafka for trending
- **A/B testing:** Multiple trie versions for experimentation

8. Design a video streaming platform like YouTube. Discuss video processing, CDN strategy, adaptive bitrate streaming, and storage optimization.

System Components

- **Upload Service:** Handle video uploads (chunked, resumable)
- **Transcoding Service:** Convert to multiple formats/resolutions
- **Storage:** Object storage (S3/GCS) for raw and processed videos
- **CDN:** CloudFront/Akamai for content delivery
- **Metadata Service:** Video info, thumbnails, captions
- **Streaming Service:** Serve video segments (HLS/DASH)

Video Processing Pipeline

- **Upload:** Chunk upload to S3 with multipart API
- **Transcoding:** FFmpeg workers generate multiple bitrates (360p, 720p, 1080p, 4K)
- **Thumbnail:** Extract keyframes at intervals
- **Encryption:** DRM for premium content
- **Notification:** Update status via webhook/queue

Adaptive Bitrate Streaming

```
import subprocess

def transcode_video(input_file, output_dir):
    resolutions = [
        ('640x360', '800k'),
        ('1280x720', '2500k'),
        ('1920x1080', '5000k')
    ]
    for res, bitrate in resolutions:
        output = f'{output_dir}/{res}.m3u8'
        subprocess.run([
            'ffmpeg', '-i', input_file,
            '-vf', f'scale={res}',
            '-b:v', bitrate,
            '-hls_time', '10',
            '-hls_list_size', '0',
            output
        ])
```

CDN Strategy

- **Origin servers:** Store master copies in S3
- **Edge caching:** Cache popular videos at edge locations
- **Cache invalidation:** Versioned URLs or purge API
- **Geographic routing:** Route users to nearest edge
- **Cost optimization:** Use tiered storage (S3 Glacier for old videos)

Streaming Protocol

- **HLS (HTTP Live Streaming):** Apple standard, wide support
- **DASH:** Industry standard, more flexible
- **Segment duration:** 6-10 seconds for balance
- **Manifest file:** Lists available bitrates and segments

Storage Optimization

- **Compression:** H.264/H.265 codecs
- **Deduplication:** Hash-based detection of duplicate uploads
- **Lifecycle policies:** Move to cold storage after inactivity
- **Partial uploads:** Resume failed uploads

9. Design a distributed logging and monitoring system for microservices. How would you collect, aggregate, store, and query logs at petabyte scale?

Architecture Components

- **Log Agents:** Fluentd/Filebeat on each service
- **Message Queue:** Kafka for buffering and decoupling
- **Stream Processing:** Flink/Spark for aggregation and filtering
- **Storage:** Elasticsearch for search, S3/HDFS for archival
- **Visualization:** Kibana/Grafana for dashboards
- **Alerting:** Prometheus/AlertManager for anomalies

Log Collection

```
import logging
import json
```

```
class StructuredLogger:
    def __init__(self, service_name):
        self.service = service_name
        self.logger = logging.getLogger(service_name)

    def log(self, level, message, **context):
        log_entry = {
            'timestamp': time.time(),
            'service': self.service,
            'level': level,
            'message': message,
            **context
        }
        self.logger.log(level, json.dumps(log_entry))
```

Data Pipeline

- **Collection:** Agents tail log files or read stdout
- **Enrichment:** Add metadata (host, region, environment)
- **Buffering:** Kafka topics partitioned by service/severity
- **Processing:** Filter, parse, aggregate in real-time
- **Indexing:** Write to Elasticsearch with daily indices
- **Archival:** Compress and move to S3 after retention period

Schema Design

- **Structured logging:** JSON format with consistent fields
- **Trace ID:** Correlation across microservices
- **Standard fields:** timestamp, level, service, message, context
- **Index strategy:** Time-based indices (logs-2024-01-15)

Query Optimization

- **Index patterns:** Query across multiple time-based indices
- **Aggregations:** Pre-compute metrics (error rates, latencies)
- **Sampling:** Store only sample of debug logs
- **Hot/warm/cold:** Tier storage by age and access pattern

Scalability Considerations

- **Partitioning:** Kafka partitions for parallelism
- **Backpressure:** Handle spikes with buffering
- **Retention:** Delete old indices automatically
- **Compression:** Reduce storage costs (gzip/snappy)
- **Sharding:** Distribute Elasticsearch across nodes

Monitoring and Alerting

- Track error rate spikes
- Monitor log pipeline health
- Alert on anomalies (ML-based)
- SLA dashboards for key metrics

10. Design an e-commerce inventory management system. How would you handle concurrent updates, prevent overselling, and maintain consistency across distributed warehouses?

Core Challenges

- **Race conditions:** Multiple users buying last item
- **Distributed inventory:** Stock across multiple warehouses
- **Consistency:** Ensure accurate stock levels
- **Reservations:** Hold inventory during checkout
- **Performance:** Handle flash sales (thousands of requests/sec)

Database Design

- **Inventory table:** (product_id, warehouse_id, quantity, reserved, version)
- **Reservation table:** (reservation_id, product_id, quantity, user_id, expires_at)
- **Order table:** (order_id, product_id, quantity, status)

Pessimistic Locking Approach

```
def reserve_inventory(product_id, quantity):
    with db.transaction():
        inventory = db.execute(
            'SELECT * FROM inventory WHERE product_id = %s FOR UPDATE',
            product_id
        ).fetchone()

        available = inventory.quantity - inventory.reserved
        if available >= quantity:
            db.execute(
                'UPDATE inventory SET reserved = reserved + %s WHERE product_id = %s',
                quantity, product_id
            )
            return True
        return False
```

Optimistic Locking Approach

```
def reserve_with_version(product_id, quantity):
    while True:
        inv = db.get_inventory(product_id)
        if inv.quantity - inv.reserved < quantity:
            return False

        updated = db.execute(
            'UPDATE inventory SET reserved = reserved + %s WHERE product_id = %s AND version = %s',
            quantity, product_id, inv.version
        )
        if updated.rowcount > 0:
            return True
```

Distributed Inventory Strategy

- **Routing:** Check nearest warehouse first
- **Fallback:** Try other warehouses if local stock insufficient
- **Split orders:** Fulfill from multiple warehouses
- **Real-time sync:** Update inventory across all locations

Reservation System

- **Temporary hold:** Reserve inventory for 10-15 minutes during checkout
- **TTL expiration:** Auto-release expired reservations
- **Background job:** Cleanup expired reservations periodically
- **Idempotency:** Prevent duplicate reservations

Flash Sale Optimization

- **Queue system:** Kafka/Redis queue to serialize requests
- **Cache:** Cache inventory count in Redis
- **Rate limiting:** Prevent bot abuse
- **Eventual consistency:** Accept slight overselling, handle refunds

CAP Theorem Trade-offs

- Prioritize **Consistency** over Availability for inventory
- Use distributed transactions (2PC) for critical operations
- Accept partition tolerance with compensation mechanisms

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a function to flatten a nested list of arbitrary depth without using recursion.

Solution using iterative approach with a stack:

```
def flatten(nested_list):
    stack = list(nested_list)
    result = []
    while stack:
        item = stack.pop()
        if isinstance(item, list):
            stack.extend(item)
        else:
            result.append(item)
    return result[::-1]
```

Key Points:

- Uses a stack to process elements iteratively
- Checks type using isinstance() for list detection
- Reverses result since stack is LIFO
- Time complexity: O(n) where n is total elements

2. How do you reverse a string in-place in Python, and why is it challenging?

Challenge: Strings in Python are **immutable**, so true in-place reversal is impossible. You must convert to a mutable type.

```
def reverse_string(s):
    chars = list(s)
    left, right = 0, len(chars) - 1
    while left < right:
        chars[left], chars[right] = chars[right], chars[left]
        left += 1
        right -= 1
    return ''.join(chars)
```

Alternative (Pythonic):

```
reversed_str = s[::-1]
```

Key Points:

- String immutability requires conversion to list
- Two-pointer technique for manual reversal
- Slicing with [::-1] is idiomatic but creates new object

3. Write an efficient function to check if a string is a palindrome, ignoring spaces and case.

Optimized solution with two-pointer technique:

```
def is_palindrome(s):
    cleaned = ''.join(c.lower() for c in s if c.isalnum())
    left, right = 0, len(cleaned) - 1
    while left < right:
        if cleaned[left] != cleaned[right]:
            return False
        left += 1
        right -= 1
    return True
```

Key Points:

- Use isalnum() to filter alphanumeric characters
- Two-pointer approach avoids creating reversed copy
- Time: O(n), Space: O(n) for cleaned string
- Can optimize to O(1) space by checking original string indices

4. What are the best practices for exception handling in Python? Provide an example of proper exception hierarchy usage.

Best Practices:

- Catch specific exceptions, not bare except
- Use finally for cleanup operations
- Leverage exception hierarchy for granular handling
- Use else clause for code that runs only if no exception occurs

Example:

```
try:
    result = risky_operation()
except ValueError as e:
    handle_value_error(e)
except (IOError, OSError) as e:
    handle_io_error(e)
else:
    process_success(result)
finally:
    cleanup_resources()
```

Advanced: Create custom exception hierarchies inheriting from appropriate base exceptions for domain-specific error handling.

5. Explain monkey patching in Python. When is it appropriate and what are the risks?

Monkey Patching: Dynamically modifying or extending code at runtime by changing attributes/methods of classes or modules.

```
import datetime
original_now = datetime.datetime.now
def mock_now():
    return datetime.datetime(2024, 1, 1)
datetime.datetime.now = mock_now
# Now all calls return fixed date
```

Appropriate Use Cases:

- Testing: Mocking dependencies or time-dependent code
- Hot-fixing third-party libraries temporarily
- Adding functionality to sealed classes

Risks:

- Makes code harder to understand and maintain
- Can break if library internals change
- Thread-safety issues in concurrent environments
- Better alternatives: dependency injection, wrapper classes

6. How do you profile memory usage in Python applications? Provide practical examples.

Memory Profiling Tools:

1. memory_profiler

```
from memory_profiler import profile
@profile
def memory_intensive():
    large_list = [i for i in range(10**6)]
    return sum(large_list)
```

2. tracemalloc (built-in)

```
import tracemalloc
tracemalloc.start()
# Your code here
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')
for stat in top_stats[:3]:
    print(stat)
```

Best Practices:

- Use tracemalloc for production (no dependencies)
- Profile incrementally to identify memory leaks
- Monitor peak memory with resource module
- Use gc module to track reference cycles

7. What debugging tools and techniques do you use beyond print statements? Explain pdb usage.

Advanced Debugging Tools:

- **pdb:** Python debugger (built-in)
- **ipdb:** IPython-enhanced debugger
- **pudb:** Full-screen console debugger
- **IDE debuggers:** PyCharm, VS Code with breakpoints

pdb Usage:

```
import pdb
def buggy_function(x, y):
    result = x + y
    pdb.set_trace()  # Breakpoint
    return result * 2
```

Common pdb Commands:

- **n:** Next line
- **s:** Step into function
- **c:** Continue execution
- **p variable:** Print variable value
- **l:** List source code
- **bt:** Backtrace (call stack)

Python 3.7+: Use breakpoint() instead of pdb.set_trace()

8. Write a function to find the first non-repeating character in a string efficiently.

Optimal solution using hash map:

```
from collections import Counter
def first_non_repeating(s):
    counts = Counter(s)
    for char in s:
        if counts[char] == 1:
            return char
    return None
```

Alternative with OrderedDict (manual approach):

```
def first_non_repeating_v2(s):
    char_count = {}
    for char in s:
        char_count[char] = char_count.get(char, 0) + 1
    for char in s:
        if char_count[char] == 1:
            return char
    return None
```

Complexity:

- Time: O(n) - two passes through string
- Space: O(k) where k is unique characters

9. Explain the difference between is and == operators. When would they produce different results?

Key Difference:

- **==** checks value equality (calls __eq__ method)
- **is** checks identity (same object in memory)

Examples showing differences:

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a
print(a == b)  # True (same values)
print(a is b)  # False (different objects)
print(a is c)  # True (same object)
x = 256
y = 256
print(x is y)  # True (integer caching)
z = 257
w = 257
print(z is w)  # False (outside cache range)
```

Important Notes:

- Python caches small integers (-5 to 256)
- String interning affects string identity
- Always use **is** for None checks: if x is None
- Use **==** for value comparisons

10. How do you implement a custom context manager? Provide both class-based and decorator-based approaches.

Class-based approach:

```
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file
    def __exit__(self, exc_type, exc_val, exc_tb):
        self.file.close()
        return False
```

Decorator-based (contextlib):

```
from contextlib import contextmanager
@contextmanager
def file_manager(filename, mode):
    f = open(filename, mode)
    try:
        yield f
    finally:
        f.close()
```

Usage:

```
with file_manager('data.txt', 'r') as f:
    content = f.read()
```

Key Points:

- __enter__ returns resource, __exit__ handles cleanup
- __exit__ receives exception info if error occurs
- Return True from __exit__ to suppress exceptions
- contextmanager decorator simplifies implementation

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you had to optimize a Python application that was experiencing performance issues.

Situation: Our Django-based API was experiencing slow response times (3-5 seconds) during peak hours, affecting 10,000+ daily users.

Task: I was tasked with identifying bottlenecks and reducing response time to under 500ms within two weeks.

Action: I profiled the application using cProfile and Django Debug Toolbar, discovered N+1 query problems and inefficient database queries. I implemented select_related() and prefetch_related() for ORM optimization, added Redis caching for frequently accessed data, and introduced database indexing on foreign keys. I also refactored a computationally expensive algorithm using NumPy for vectorized operations.

Result: Response times dropped to an average of 300ms (90% improvement), server costs decreased by 40% due to reduced CPU usage, and user satisfaction scores improved by 25%.

2. Describe a situation where you had to debug a complex issue in production. How did you approach it?

Situation: A critical payment processing service started failing intermittently in production, causing transaction failures for approximately 15% of users with no clear error patterns.

Task: I needed to identify and resolve the issue quickly while maintaining service availability and preventing data loss.

Action: I immediately implemented comprehensive logging using Python's logging module with structured JSON logs, added distributed tracing with OpenTelemetry to track requests across microservices, and analyzed logs using ELK stack. I discovered a race condition in our async payment validation logic caused by improper use of asyncio locks. I implemented proper synchronization using asyncio.Lock() and added idempotency keys to prevent duplicate processing.

Result: Transaction failure rate dropped to 0.2%, we prevented $50K in potential lost revenue, and established better monitoring practices that reduced mean time to detection (MTTD) by 60% for future incidents.

3. Give an example of when you had to mentor or lead other developers on a Python project.

Situation: I was assigned to lead a team of three junior developers on a new microservices migration project, and they had limited experience with Python best practices and modern architectural patterns.

Task: My goal was to deliver the project on time while upskilling the team on Python development, testing practices, and system design.

Action: I established weekly code review sessions focusing on Pythonic code patterns, created internal documentation on our coding standards (PEP 8, type hints, docstrings), implemented pair programming for complex features, and introduced pytest with emphasis on test-driven development. I also created reusable templates and shared resources on design patterns like dependency injection and repository pattern.

Result: The team successfully delivered the project two weeks ahead of schedule, code quality metrics improved by 40% (measured by reduced code review iterations), all team members became proficient in writing unit and integration tests (achieving 85% coverage), and two team members were promoted within six months.

4. Tell me about a time when you disagreed with a technical decision. How did you handle it?

Situation: During architecture planning for a new data pipeline, the team lead proposed using a custom-built solution in Python instead of adopting Apache Airflow, citing team familiarity concerns.

Task: I believed this decision would create technical debt and maintenance overhead, but needed to present my concerns constructively without undermining leadership.

Action: I prepared a detailed technical comparison document highlighting maintenance costs, scalability concerns, and long-term support implications. I created a proof-of-concept with Airflow demonstrating how quickly our team could adapt, estimated 200+ hours of development time we'd save, and proposed a phased adoption plan with training sessions. I presented this in a collaborative manner, acknowledging valid concerns about learning curve.

Result: After reviewing the evidence, the team agreed to adopt Airflow. The decision saved an estimated 300 development hours, we leveraged built-in features like retry logic and monitoring, and the team became proficient within three weeks. The lead later thanked me for the thorough analysis and collaborative approach.

5. Describe a challenging bug you encountered in Python and how you resolved it.

Situation: Our machine learning pipeline started producing inconsistent predictions in production despite identical code working perfectly in development and staging environments.

Task: I needed to identify why the same Python code and model produced different outputs across environments, as this was affecting business-critical recommendations.

Action: I systematically compared environments including Python versions, library versions, and system configurations. I discovered the issue was caused by non-deterministic behavior in dictionary ordering (pre-Python 3.7 on production) affecting feature engineering. I implemented OrderedDict for consistent ordering, added explicit random seed setting for NumPy and TensorFlow, and created comprehensive environment validation scripts using pip freeze and conda list. I also containerized the application using Docker to ensure environment consistency.

Result: Predictions became 100% reproducible across all environments, we prevented potential revenue loss from incorrect recommendations, established containerization as a standard practice (reducing environment-related issues by 80%), and created documentation on deterministic ML pipelines.

6. Tell me about a time when you had to make a trade-off between code quality and delivery speed.

Situation: We had a critical client deadline for a new API feature in two weeks, but implementing it with full test coverage, documentation, and optimal architecture would require four weeks.

Task: I needed to deliver a working solution by the deadline while minimizing technical debt and ensuring maintainability.

Action: I proposed a phased approach: implement core functionality with essential error handling and input validation for the deadline, maintain critical unit tests for business logic (60% coverage), use type hints for self-documenting code, and explicitly document technical debt items in JIRA with time estimates. I negotiated with stakeholders to allocate two weeks post-launch for refactoring, adding comprehensive tests, and performance optimization. I ensured the initial code was modular to facilitate future improvements.

Result: We met the client deadline successfully, the feature worked reliably with zero critical bugs in production, we completed the technical debt work as planned achieving 90% test coverage, and established this phased approach as a template for future urgent deliverables. The client renewed their contract citing our responsiveness.

7. Describe a situation where you had to learn a new Python library or framework quickly for a project.

Situation: Our team was tasked with building a real-time data processing system, and stakeholders decided on using FastAPI and Apache Kafka, neither of which I had production experience with, and we had only three weeks until launch.

Task: I needed to become proficient enough to architect and implement a production-grade solution using these technologies within the tight timeline.

Action: I created a structured learning plan: spent the first three days on official documentation and tutorials, built a small proof-of-concept replicating our use case, studied production-ready examples on GitHub, joined relevant Discord communities and Stack Overflow for quick problem-solving, and consulted with colleagues who had experience. I documented learnings and gotchas in a team wiki, implemented the solution incrementally with code reviews, and wrote comprehensive tests to catch issues early.

Result: Successfully delivered the system on time with 95% uptime in the first month, the solution handled 10,000+ messages per second efficiently, my documentation helped onboard two additional team members in half the expected time, and I became the team's go-to resource for FastAPI, later conducting internal training sessions.

8. Tell me about a time when you improved the development workflow or practices for your Python team.

Situation: Our Python team was experiencing frequent production bugs, inconsistent code styles across developers, and lengthy code review cycles averaging 3-4 days.

Task: I volunteered to improve our development practices to reduce bugs, accelerate reviews, and improve code consistency.

Action: I introduced and configured pre-commit hooks with Black for formatting, Flake8 for linting, and mypy for type checking. I set up GitHub Actions CI/CD pipeline running automated tests and code quality checks on every PR. I created a comprehensive style guide based on PEP 8 with team-specific conventions, implemented mandatory type hints policy for new code, and introduced pull request templates with checklists. I also organized weekly knowledge-sharing sessions on Python best practices and organized the codebase structure using clear module separation.

Result: Code review time decreased from 3-4 days to under 24 hours (70% improvement), production bugs reduced by 45% over three months, code style became consistent across the team eliminating style-related review comments, and developer satisfaction improved as evidenced by retrospective feedback. The practices were adopted by two other teams in the organization.

9. Describe a time when you had to refactor legacy Python code. What was your approach?

Situation: I inherited a 15,000-line Python 2.7 monolithic application with no tests, poor documentation, and multiple critical bugs reported monthly. The application was business-critical but becoming unmaintainable.

Task: I was tasked with modernizing the codebase to Python 3, improving maintainability, and reducing bug frequency without disrupting the production service.

Action: I developed a systematic refactoring strategy: first, added comprehensive integration tests to capture existing behavior, used 2to3 tool and six library for initial Python 3 migration, then incrementally refactored modules starting with the most bug-prone areas. I extracted business logic into separate services, introduced dependency injection for testability, added unit tests achieving 80% coverage, and documented architectural decisions using ADRs (Architecture Decision Records). I deployed changes incrementally using feature flags to minimize risk.

Result: Successfully migrated to Python 3.9 with zero downtime, reduced production bugs by 65% over six months, decreased average bug fix time from 2 days to 4 hours, improved code maintainability score from D to B (CodeClimate), and the refactored architecture enabled adding new features 3x faster than before.

10. Tell me about a time when you had to balance multiple competing priorities in a Python project.

Situation: I was simultaneously responsible for: delivering a new feature for a major client (2-week deadline), fixing a critical security vulnerability in our authentication system, mentoring two junior developers, and addressing technical debt in our data processing pipeline.

Task: I needed to manage all responsibilities effectively without compromising quality or missing deadlines, while maintaining team morale.

Action: I immediately assessed and prioritized based on impact and urgency: addressed the security vulnerability first (completed in 1 day with thorough testing), delegated portions of the new feature to junior developers with clear specifications and daily check-ins (using this as a mentoring opportunity), dedicated focused time blocks for code reviews and mentoring (30 minutes daily), and negotiated with stakeholders to defer non-critical technical debt work by two weeks. I used time-blocking techniques, communicated status updates proactively, and worked one weekend to ensure the client feature met quality standards.

Result: Delivered the client feature on time with positive feedback, the security vulnerability was patched within SLA, junior developers successfully completed their assigned tasks and grew their skills, technical debt was addressed in the following sprint, and I received recognition for effective prioritization and stakeholder management. The experience helped me develop a priority framework I now use regularly.