BRP DEVELOPERS / FREE LEARNING LIBRARY

Python Interview Deep Dive

Advanced edition ? BRP developers free companion guide

Python interviews | Advanced

Original educational companion guide. No affiliation with any interview platform or certification provider.

This guide focuses on Python's data model, reliable abstractions, asynchronous work, and evidence-based performance. Examples are original. The goal is not to memorize implementation trivia; it is to explain trade-offs and build code whose behavior can be tested.

The data model is a protocol system

Python syntax often delegates to special methods. `len(value)` invokes `value.__len__()`, iteration uses the iterator protocol, and `a + b` may call `a.__add__(b)` before considering the reflected operation on `b`.

A custom value object should implement the smallest protocol needed by callers. For addition, return `NotImplemented` for unsupported operands rather than raising `TypeError` immediately; Python can then try the reflected method:

```
class Distance:
    def __init__(self, metres):
        self.metres = float(metres)

    def __add__(self, other):
        if not isinstance(other, Distance):
            return NotImplemented
        return Distance(self.metres + other.metres)
```

If defining equality, consider hashing and ordering together. Equal values need equal hashes. Mutable objects whose equality changes over time should usually remain unhashable. Use `dataclass(frozen=True)` for simple immutable records; use an explicit class when validation or behavior deserves a named abstraction.

Descriptors, properties, and validation boundaries

A descriptor implements `__get__`, `__set__`, or `__delete__` and can manage attribute access. `property` is the familiar built-in descriptor. A property is useful when it preserves a simple attribute interface while enforcing an invariant:

```
class Percentage:
    def __init__(self, value):
        self.value = value

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, number):
        if not 0 <= number <= 100:
            raise ValueError("percentage must be from 0 to 100")
        self._value = number
```

Do not use descriptors merely to make a field look clever. Prefer explicit validation at a boundary when the rule applies only once. A reusable descriptor can be justified when the same behavior is shared consistently across many classes.

Decorators should preserve the callable contract

A decorator wraps a callable. `functools.wraps` preserves metadata such as the function name and docstring, which helps introspection and tooling:

```
from functools import wraps

def announce(function):
    @wraps(function)
```

```
    def wrapper(*args, **kwargs):
        return function(*args, **kwargs)
    return wrapper
```

A cache is only correct when the function is referentially transparent for its cache key and results are safe to reuse. Caching a function that depends on mutable global state, current time, user identity, or a changing database can return stale or cross-user data. Define invalidation and memory bounds as part of the design, not as an afterthought.

Context managers encode ownership

A context manager scopes acquisition and release. `contextlib.contextmanager` can express a small resource lifecycle without a class:

```
from contextlib import contextmanager

@contextmanager
def managed_connection(connect):
    connection = connect()
    try:
        yield connection
    finally:
        connection.close()
```

The `finally` block runs if the body raises. In production, prefer the library's own context manager where it exists; a database connection may need commit/rollback semantics, not merely `close()`. Explain ownership: who creates a resource, who closes it, and whether exceptions are propagated.

Concurrency is about the workload

Threads are useful for overlapping blocking I/O, but shared mutable state needs synchronization. Processes can use multiple CPU cores for CPU-bound Python work, at the cost of serialization and process-management overhead. `asyncio` is useful when many tasks spend time awaiting nonblocking I/O; it does not automatically make CPU-heavy code faster.

In Python 3.11 and later, `asyncio.TaskGroup` provides structured concurrency: child tasks belong to a scope, and failures are propagated as the group exits. Cancellation is part of the API contract. Coroutines should clean up in `finally` and re-raise `CancelledError` after cleanup rather than swallowing cancellation.

```
import asyncio

async def load_pair(fetch, left, right):
    async with asyncio.TaskGroup() as group:
        a = group.create_task(fetch(left))
        b = group.create_task(fetch(right))
    return a.result(), b.result()
```

Discuss limits, timeouts, retries, and partial failure. Parallel requests can overload a downstream service unless concurrency is bounded. For a CPU-bound algorithm, first measure; an executor or another implementation may be more appropriate than adding `async` syntax.

Measure performance before optimizing

Start with a representative input and a correctness test. Use `timeit` for focused microbenchmarks, `cProfile` for call-level profiling, and `tracemalloc` for Python allocation traces. A microbenchmark can mislead when it excludes I/O, warm-up, realistic data distributions, or interpreter variation.

The usual optimization order is: choose a better algorithm, avoid duplicated work, reduce unnecessary allocations, then tune a measured hot path. Replacing readable code with a lower-level trick is not an improvement if the bottleneck lies elsewhere. State the before-and-after measurement and keep a regression test.

API and architecture questions

When designing a service boundary, validate untrusted input at the edge, keep domain rules in one place, and make side effects visible. Separate pure transformations from I/O where this makes tests simpler. Use a timeout for remote calls; retries need a policy and are unsafe for non-idempotent operations without an idempotency key.

A useful failure model distinguishes invalid input, not-found state, dependency failure, and unexpected defects. Log enough context to diagnose a failure, but do not log credentials, session tokens, or sensitive personal data. An interview answer should connect the design to its operational behavior: latency, consistency, recovery, observability, and cost.

Advanced practice prompts

1. A custom object is used as a key, then one of its fields changes. Explain the lookup failure and redesign the key as an immutable value.
2. A decorator caches results from a function that reads a feature flag. Identify the stale-result risk and propose explicit versioning or cache invalidation.
3. Three independent network calls must all finish, but one can fail. Compare TaskGroup behavior with returning partial results, and define cancellation and timeout policy.
4. A function is slow on a 10 MB input. Describe how you would profile it before choosing an optimization.

Answer framework

For each prompt, state the invariant, identify the failure mode, and name a test that would prove the fix. For concurrency and caching, mention lifecycle and invalidation. For performance, provide evidence rather than a guess. Advanced Python is most useful when its protocol behavior is made explicit and its trade-offs are measured.

More free guides: brpdevs.online/learn/