BRP DEVELOPERS / FREE LEARNING LIBRARY

Python Interview Field Guide

Mid-level edition ? BRP developers free companion guide

Python interviews | Intermediate

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

A practical guide to explaining Python behavior, choosing clear implementations, and testing edge cases in an interview. Examples use modern Python 3 and the standard library. They are original educational material, not a promise of interview outcomes.

A repeatable way to answer

Before writing code, restate the input, output, constraints, and failure cases. Ask whether inputs can be empty, repeated, unsorted, very large, or invalid. Offer a simple correct approach, then improve it only when constraints require it. State time and extra-space complexity separately. After coding, test one ordinary case, one boundary case, and one adversarial case.

A useful explanation sounds like this: ?I will count values in one pass using a dictionary. The expected time is O(n), with O(k) extra space for k distinct values. I will check the empty input and duplicates explicitly.? Naming the invariant makes the code easier to review.

Names, objects, and mutation

Python variables are names bound to objects. Assignment does not copy an object. Two names can therefore refer to the same mutable list:

```
first = [1, 2]
second = first
second.append(3)
assert first == [1, 2, 3]
```

A shallow copy creates a new outer container but keeps references to nested objects. Use `first.copy()` or `list(first)` for a shallow list copy. Use `copy.deepcopy` only when recursive copying is actually needed; it can be expensive and may not fit objects with external resources.

Function parameters are also names. A function that mutates a passed list changes the caller-visible object. A function that rebinds its local parameter does not rebind the caller's name:

```
def add_marker(items):
    items.append("done")       # mutation is visible to caller

def replace_local(items):
    items = ["new"]            # only changes this local name
```

For interview questions about ?pass by value or reference,? a precise description is more useful: Python passes object references by assignment; mutations can be observed through aliases, while local rebinding is local.

Mutable defaults and explicit state

Default expressions are evaluated once, when the function is defined. A list default is therefore shared across calls:

```
def collect(value, items=[]):
    items.append(value)
    return items
```

Use `None` as a sentinel when each call should create fresh state:

```
def collect(value, items=None):
    if items is None:
        items = []
    items.append(value)
    return items
```

```
assert collect("a") == ["a"]
assert collect("b") == ["b"]
```

This pattern also distinguishes ?argument omitted? from an explicitly supplied empty list. If that distinction matters, document it. For keyword-only options and complex configuration, a small immutable configuration object can be clearer than many defaults.

Iterables, iterators, and generators

An iterable can produce an iterator; an iterator remembers progress and is usually consumed. A list can be iterated many times. A generator expression computes values lazily and is generally one-shot:

```
squares = (number * number for number in range(4))
assert list(squares) == [0, 1, 4, 9]
assert list(squares) == []
```

Laziness avoids storing every intermediate value, but it does not make the total work disappear. If a consumer needs two passes, materialize the values or generate them again. When adapting a stream, yield one item at a time:

```
def nonblank_lines(lines):
    for line in lines:
        cleaned = line.strip()
        if cleaned:
            yield cleaned
```

A generator retains its local state between yields. This is useful for pipelines, but it also means exceptions can occur during iteration rather than at generator creation. Put resource cleanup in a context manager rather than relying on a generator being exhausted.

Equality, hashing, and data models

`is` asks whether two names refer to the same object. == asks whether objects compare equal. Use equality for values and identity for sentinels such as `None`:

```
if result is None:
    ...
```

Dictionary keys and set members must be hashable. Equal objects must have equal hashes. If an object's equality depends on mutable fields, making it hashable is dangerous because mutation can make it unreachable in a set or dictionary. A frozen dataclass is often suitable for value records:

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Point:
    x: int
    y: int
```

Use `order=True` only when the generated field-by-field ordering matches the domain. For custom rich comparisons, return `NotImplemented` for unsupported types so Python can try the other operand's implementation. Avoid writing equality based on a formatted representation; compare the actual fields.

Exceptions and context managers

Catch the narrowest exception you can handle. A broad `except Exception` can hide programming errors. Use `try/finally` or a context manager when a resource must be released even after failure:

```
with open("notes.txt", encoding="utf-8") as handle:
    first_line = handle.readline()
```

A context manager's `__exit__` method may suppress an exception only by returning true. Suppression should be deliberate. When translating an exception into a domain-specific error, preserve the cause with `raise NewError(...) from error`.

Tests that expose behavior

Test observable results, not private implementation details. `unittest` and `pytest` both support arrange-act-assert structure. For a function that groups values, cover empty input, one group, repeated values, and ordering requirements. For a parser, include malformed input and Unicode. For a cache, distinguish a missing key from a cached false-like value.

Property-based thinking helps even without a property-testing library. If a function reverses a sequence, reversing twice should recover the original. If a function deduplicates while preserving order, every output value should occur in the input and output values should be unique. These invariants often reveal bugs missed by a single example.

Practice set with explanations

Why is `list_a = list_b` not a copy? Both names point to the same list. Mutating through either name changes the shared object. Use an explicit shallow or deep copy depending on the nested structure.

When should a generator replace a list? Use a generator when values can be consumed once and retaining every intermediate result is unnecessary. Prefer a list when random access, repeated iteration, or a stable snapshot is needed.

Why can a mutable object be a poor dictionary key? A key's hash and equality must remain stable while it is stored. Mutating fields used by either operation can violate dictionary lookup assumptions.

What does a good complexity statement include? Identify the input size, the dominant operations, the extra memory, and any assumptions such as average-case hashing. Avoid claiming a bound without naming those assumptions.

Further practice

Choose a small function from your own code. Write down its input contract, three edge cases, and a complexity bound before changing it. Then add tests for those cases and explain one alternative design. The habit of making assumptions explicit is useful well beyond interviews.

More free guides: brpdevs.online/learn/