

Python Interview Questions and Answers

Intern to senior level

Python interview questions and answers from beginner to senior: list vs tuple, is vs ==, *args and **kwargs, decorators, generators, shallow vs deep copy, mutable default arguments, the GIL, memory management, and threading vs multiprocessing vs asyncio.

11 questions Read online: <https://www.freedevtemplates.com/interview-questions/python>

1. What is the difference between a list and a tuple in Python?

Level: intern Python

A list is mutable and can be changed after creation, while a tuple is immutable; tuples are slightly faster, use less memory and can be used as dictionary keys when their items are hashable.

Use a list for a collection that grows, shrinks or is reordered, such as the items in a shopping cart. Use a tuple for a fixed group of values that belong together, such as a coordinate (x, y) or a database row. Tuples also signal to the reader that the data should not change.

Immutability applies to the tuple itself: if a tuple contains a list, that list can still be modified. A one-item tuple needs a trailing comma, (5,), because (5) is just the number 5 in parentheses.

```
items = [1, 2, 3]
items.append(4) # fine

point = (10, 20)
point[0] = 5 # TypeError: 'tuple' object does not support item assignment

locations = {(10, 20): "home"} # tuples can be dict keys
```

2. What is the difference between is and == in Python?

Level: intern Python

== checks whether two objects have equal values, while is checks whether they are the exact same object in memory.

== calls the object's `__eq__` method. is compares identity (the same as comparing `id(a) == id(b)`). Two separate lists with the same content are == but not is.

Use is for singletons, mainly None (if x is None), True and False. Do not use is to compare numbers or strings: CPython caches small integers and some strings, so the result can look right in a quick test and fail in real code.

```
a = [1, 2]
b = [1, 2]
a == b # True (same value)
a is b # False (different objects)
c = a
c is a # True

if result is None:
    ...
```

3. What are *args and **kwargs in Python?

Level: junior Python

***args collects extra positional arguments into a tuple and **kwargs collects extra keyword arguments into a dictionary, which lets a function accept a variable number of arguments.**

The names args and kwargs are conventions; the stars are what matter. They are useful for wrappers and decorators that must pass any arguments through to another function, and for APIs that accept optional settings.

The same stars work at the call site to unpack: `print(*numbers)` spreads a list into separate arguments and `func(**options)` spreads a dictionary into keyword arguments. The order in a signature is: normal parameters, *args, keyword-only parameters, **kwargs.

```
def report(title, *args, **kwargs):
    print(title, args, kwargs)

report("Sales", 1, 2, region="EU", year=2026)
# Sales (1, 2) {'region': 'EU', 'year': 2026}

nums = [3, 4]
opts = {"sep": "-"}
print(*nums, **opts) # 3-4
```

4. Why should you not use a mutable default argument in Python?

Level: junior Python

Default argument values are evaluated once when the function is defined, so a default list or dictionary is shared between every call that does not pass its own, causing data to leak between calls.

This is one of the most common Python gotchas. The fix is to use None as the default and create a new object inside the function.

```
def add_item(item, items=[]): # bug: one shared list
    items.append(item)
    return items

add_item("a") # ['a']
add_item("b") # ['a', 'b'] <- surprise

def add_item(item, items=None): # correct
    if items is None:
        items = []
    items.append(item)
    return items
```

5. What is the difference between a shallow copy and a deep copy in Python?

Level: junior Python

A shallow copy creates a new container but reuses the same nested objects, while a deep copy recursively copies everything so the new object is fully independent.

With a shallow copy (`copy.copy()`, `list.copy()`, slicing `items[:]`), changing a nested list in the copy also changes it in the original, because both point to the same inner object. `copy.deepcopy()` duplicates the nested objects as well, handling circular references, but it is slower and uses more memory.

Plain assignment `b = a` does not copy at all: both names refer to the same object.

```
import copy

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

original[0].append(99)
print(shallow[0]) # [1, 2, 99] - shared inner list
print(deep[0]) # [1, 2] - independent
```

6. What is the difference between @staticmethod and @classmethod in Python?

Level: junior Python

A classmethod receives the class as its first argument (cls) and can create or configure instances, while a staticmethod receives no automatic argument and is just a function placed inside the class namespace.

Use @classmethod for alternative constructors such as Date.from_string(...), and because it uses cls, it works correctly for subclasses. Use @staticmethod for a utility function that is related to the class but needs neither the instance nor the class. An ordinary method receives self, the instance.

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

    @classmethod
    def from_email(cls, email):
        return cls(email.split("@")[0])

    @staticmethod
    def is_valid_email(email):
        return "@" in email
```

7. What are decorators in Python and how do they work?

Level: mid Python

A decorator is a function that takes another function and returns a modified version of it, letting you add behaviour such as logging, caching or access control without changing the original code.

The @decorator syntax is shorthand for func = decorator(func). A typical decorator defines an inner wrapper function that runs code before and after calling the original function, and returns that wrapper. Use functools.wraps so the wrapped function keeps its name and docstring.

Decorators rely on functions being first-class objects and on closures. Common built-in ones include @property, @staticmethod, @classmethod and functools.lru_cache. Frameworks use them heavily, for example @app.route("/") in Flask.

```
import functools, time

def timed(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.perf_counter() - start:.3f}s")
        return result
    return wrapper

@timed
def load_data():
    ...
```

8. What are generators and the yield keyword in Python?

Level: mid Python

A generator is a function that uses yield to produce values one at a time and remembers its state between calls, so it can process large or infinite sequences without holding everything in memory.

Calling a generator function does not run it; it returns an iterator. Each next() call resumes the function until the next yield. A list of ten million numbers takes hundreds of megabytes, while a generator producing them one by one uses almost none.

A generator expression, (x * x for x in range(10**7)), is the lazy counterpart of a list comprehension. Generators can only be consumed once. They are ideal for reading big files line by line, streaming API pages and building data pipelines.

Key points

yield pauses the function and keeps local state

Lazy evaluation saves memory

A generator can be iterated only once

```
def read_large_file(path):
    with open(path) as f:
        for line in f:
            yield line.strip()

for row in read_large_file("big.csv"):
    process(row) # one line in memory at a time

total = sum(x * x for x in range(1_000_000)) # no list created
```

9. How does Python manage memory and garbage collection?

Level: mid Python

CPython frees objects mainly through reference counting, deleting an object as soon as nothing refers to it, and uses a cyclic garbage collector to clean up groups of objects that reference each other.

Every object keeps a count of references to it. When the count reaches zero its memory is released immediately. This makes cleanup predictable, but reference counting cannot free cycles, such as two objects that point at each other. The gc module's generational collector periodically finds and frees those.

Memory leaks in Python are usually objects kept alive by long-lived references: global lists or dictionaries, caches without limits, or closures. Use weakref for caches that should not keep objects alive, __slots__ to shrink many small objects, generators to stream data, and tools such as tracemalloc to see what is allocating memory.

10. What is the GIL in Python and how does it affect multithreading?

Level: senior Python

The Global Interpreter Lock (GIL) in CPython lets only one thread execute Python bytecode at a time, so threads do not speed up CPU-bound code but still help with I/O-bound work.

The GIL exists to keep CPython's memory management (reference counting) simple and safe. A thread releases it while waiting on I/O such as network calls and file reads, so many I/O-bound threads can overlap their waiting. For pure-Python CPU-bound loops, threads take turns and the program is no faster, sometimes slower.

To use multiple cores, use multiprocessing or `concurrent.futures.ProcessPoolExecutor`, which run separate interpreters, or push the heavy work into C extensions such as NumPy that release the GIL. Recent Python versions have an experimental free-threaded build that removes the GIL, but the default interpreter still has it.

Key points

One thread runs Python bytecode at a time in CPython

Threads are fine for I/O-bound work

Use processes for CPU-bound work

11. What is the difference between threading, multiprocessing and asyncio in Python?

Level: senior Python

Threading runs concurrent threads that share memory and suit blocking I/O, multiprocessing runs separate processes to use multiple CPU cores for CPU-bound work, and asyncio runs many non-blocking tasks cooperatively on one thread for high-volume I/O.

Choose by the type of bottleneck. For CPU-bound work such as image processing or number crunching, use multiprocessing so each worker has its own GIL, and accept the cost of starting processes and passing data between them. For I/O-bound work with libraries that block, such as requests, threads are simple and effective.

For thousands of concurrent network connections, asyncio with `async/await` has the lowest overhead because tasks yield control at each `await`, but every library in the path must be `async`-aware, and one blocking call stalls the whole event loop. Many production systems combine them, for example an `async` web server that sends CPU-heavy jobs to a process pool or task queue.

Key points

CPU-bound: multiprocessing

Blocking I/O: threading

Massive concurrent I/O: asyncio

```
import asyncio, httpx

async def fetch(client, url):
    r = await client.get(url)
    return r.status_code

async def main(urls):
    async with httpx.AsyncClient() as client:
        return await asyncio.gather(*(fetch(client, u) for u in urls))

asyncio.run(main(["https://example.com"] * 5))
```

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/python>