Python Interview Questions

Download PDF Print

Use the filter to quickly find topics like partitions, consumer groups, offsets, ISR, replication, retention, and more.

Filter questions (e.g., partitions, offsets, ISR, consumer

Showing 300 of 300

1. ## What is Python and what are its key features?

   Python is a high-level, interpreted, general-purpose language. It emphasizes readability, has dynamic typing and automatic memory management, and supports multiple paradigms (procedural, object-oriented, and functional). It ships with a large standard library and runs on many platforms.

2. ## How do lists differ from tuples?

   Lists are mutable, dynamic sequences (you can append/remove/modify elements). Tuples are immutable sequences (fixed once created), are slightly more memory-efficient, and can be used as dictionary keys if all contained elements are hashable.

3. ## What is the difference between a dictionary and a set?

   A dictionary stores key→value mappings with unique, hashable keys. A set stores only unique, hashable elements without associated values. Both are unordered collections with fast membership tests due to hashing.

4. ## Explain mutable vs. immutable types with examples.

   Mutable objects (list, dict, set, bytearray) can change in place. Immutable objects (int, float, bool, str, tuple, frozenset) cannot be altered after creation. Immutables are safe as dictionary keys; mutables are not (unless frozen/hashed carefully).

5. ## What are *args and **kwargs?

   args collects extra positional arguments as a tuple; **kwargs collects extra keyword arguments as a dict. They enable flexible APIs and are also used for argument unpacking when calling other functions.

6. ## What are list comprehensions and why use them?

   List comprehensions build lists in a single, readable expression (e.g., [xx for x in nums if x % 2 == 0]). They are concise and often faster than equivalent for-loops. For large data where you don’t need a list, prefer generator expressions to save memory.

7. ## How does slicing work in Python?

   Slicing uses start:stop:step on sequences (e.g., s[1:5:2]). Omitted start/stop default to the ends; negative indices count from the right. s[::-1] returns a reversed copy. Slices create new objects for lists/strings/tuples.

8. ## What is a virtual environment and why is it important?

   A virtual environment (venv) isolates a project’s Python interpreter and dependencies from the system and other projects. It prevents version conflicts and makes builds reproducible. Create with python -m venv .venv and activate it for the project.

9. ## What does if __name__ == '__main__': do?

   A module’s __name__ is set to '__main__' only when it is executed as a script. This guard lets you put runnable/demo code there while keeping import side-effects minimal when the module is imported elsewhere.

10. ## How do try/except/else/finally blocks work?

    try wraps code that may error; except handles specific exceptions; else runs only if no exception occurred; finally always runs (cleanup). Custom exceptions should subclass Exception for clarity and catchability.

11. ## What is the difference between an iterable, an iterator, and a generator?

    An iterable is any object you can loop over (has __iter__). An iterator is the object that yields values (has __next__). A generator is a special iterator created by a function with yield or a generator expression; it lazily produces values.

12. ## What are decorators and when would you use them?

    Decorators wrap callables to modify behavior without changing their code. Common uses include logging, validation, caching (@functools.lru_cache), authorization, and retry logic. Implement as functions returning wrappers; use functools.wraps to preserve metadata.

13. ## Explain instance methods vs. @classmethod vs. @staticmethod.

    Instance methods receive self and operate on object state. @classmethod receives cls and operates on class state or provides alternate constructors. @staticmethod receives neither and is just a namespaced helper function in the class.

14. ## What are data classes and why use them?

    Data classes (@dataclass) auto-generate __init__, __repr__, comparisons, and more from type-annotated fields, reducing boilerplate. Options include frozen=True (immutability), order=True (ordering), and field(default_factory=...) for mutable defaults.

15. ## Shallow copy vs. deep copy—what's the difference?

    A shallow copy (e.g., copy.copy) makes a new container but references the same nested objects. A deep copy (copy.deepcopy) recursively copies nested objects, producing an independent structure. Choose based on whether sharing inner objects is acceptable.

16. ## What is the Global Interpreter Lock (GIL) in CPython?

    The GIL is a mutex allowing only one thread to execute Python bytecode at a time. It limits true parallelism for CPU-bound code in threads but not for I/O-bound work. For CPU-bound tasks, use multiprocessing, native extensions, or other interpreters.

17. ## When should you use threading vs. multiprocessing?

    Use threading for I/O-bound tasks (network, disk) where threads can overlap waiting; the GIL isn't a big issue. Use multiprocessing for CPU-bound workloads to bypass the GIL via separate processes, at the cost of higher memory/IPC overhead.

18. ## What is asyncio and when is it appropriate?

    asyncio provides single-threaded concurrency using an event loop, coroutines (async def), and await. It excels at high-concurrency I/O (e.g., many sockets/web requests). It does not speed up CPU-bound work without offloading.

19. ## What are context managers and the purpose of with?

    Context managers manage setup/teardown reliably. The with statement calls __enter__ on entry and __exit__ on exit, even if exceptions occur. Create them with a class implementing those methods or with contextlib.contextmanager.

20. ## How does Python manage memory?

    CPython uses reference counting with a cyclic garbage collector to reclaim cycles. Objects are allocated on the heap; small integers and some strings may be interned. del lowers a reference count; __slots__ can reduce per-instance memory in classes.

21. ## How does the import system work?

    When importing, Python searches sys.modules (cache), then paths in sys.path. Packages are directories (namespace packages may omit __init__.py). A module is executed only once per process and then reused from the cache on subsequent imports.

22. ## What are type hints and how do they help?

    Type hints (PEP 484) annotate variables and function signatures (e.g., def f(x: int) -> str:). They are optional and not enforced at runtime but enable better tooling: autocompletion, static analysis (mypy, pyright), and clearer APIs.

23. ## Explain closures and the late-binding gotcha.

    A closure captures variables from an enclosing scope. In loops, lambdas/functions may capture the same variable reference (late binding). Fix by binding the current value as a default (lambda x=i: x) or using functools.partial; use nonlocal to rebind.

24. ## How does sorting work in Python?

    list.sort() and sorted() are stable and accept key= and reverse=. Provide a key function (e.g., key=str.lower) for complex sorts. The underlying algorithm is Timsort (adaptive), O(n log n) average/worst and near O(n) on nearly sorted data.

25. ## How do you test Python code?

    Use unittest in the standard library or pytest for a concise style with powerful fixtures and plugins. Organize tests per module, isolate side effects, use mocks for I/O, and run in CI. Aim for fast, deterministic tests with clear assertions.

26. ## What is PEP 8 and why does it matter?

    PEP 8 is Python's official style guide. Following it makes code consistent and readable across teams. You can enforce it with tools like flake8/pylint (linting) and black/autopep8 (formatting).

27. ## What's the difference between absolute and relative imports?

    Absolute imports use the full path from the project root (e.g., from pkg.module import A) and are clearer and recommended. Relative imports are based on the current package using dots (e.g., from .sub import A). Use relative imports for tightly coupled package internals; prefer absolute elsewhere.

28. ## How do is and == differ in Python?

    == checks value equality via __eq__, while is checks object identity (same memory object). For small integers and short strings caching may make is appear to work, but you must not rely on it for value comparisons.

29. ## Why are mutable default arguments dangerous and how do you avoid them?

    Default values are evaluated once at function definition time. Using a list/dict as a default can accumulate state across calls. Avoid by using None as the default and creating a new object inside: def f(x=None): x = [] if x is None else x.

30. ## What's the difference between __repr__ and __str__?

__repr__ returns an unambiguous developer-oriented representation (ideally valid Python to recreate the object). __str__ is a user-friendly string for display. If only one is defined, Python falls back to __repr__ for str().

31. ## How do equality, ordering, and hashing work for custom classes?

    Implement __eq__ for equality; implement ordering with functools.total_ordering or the rich methods like __lt__. If instances are hashable, implement __hash__ consistent with __eq__ (equal objects must have the same hash). Data classes can generate these with eq=True, order=True, frozen=True.

32. ## What is the Method Resolution Order (MRO) and how does super() use it?

    The MRO is the linear order Python uses to search attributes in multiple inheritance (C3 linearization). super() follows the MRO to call the next method in the chain, enabling cooperative multiple inheritance when each class calls super() appropriately.

33. ## What is the descriptor protocol and where is it used?

    Descriptors define __get__, __set__, and/or __delete__ on a class attribute to control attribute access. Built-ins like property, classmethod, staticmethod, and functools.cached_property are descriptors. They enable reusable behaviors such as validation and lazy computation.

34. ## Explain __new__ vs. __init__.

    __new__(cls, ...) creates and returns a new instance (a constructor at the allocation level) and is used for immutables or customizing creation. __init__(self, ...) initializes the already created instance. For immutable types (e.g., tuple, str), customization typically happens in __new__.

35. ## How do you create and manage context managers beyond with open()?

    Implement __enter__/__exit__ in a class or use @contextlib.contextmanager to yield a resource and handle cleanup in a try/finally. For multiple resources, contextlib.ExitStack helps compose them dynamically.

36. ## Why use pathlib over os.path?

    pathlib provides an object-oriented, cross-platform API for paths: / operator for joining, methods like read_text(), iterdir(), and easy conversions. It improves readability and reduces string concatenation mistakes.

37. ## What are best practices for file I/O and encodings?

    Use with to ensure proper closing. Specify encodings explicitly (e.g., encoding='utf-8'). For large files, process in chunks/stream lines. Be mindful of newline handling (newline='') when working with CSV via csv module.

38. ## How should you configure logging in Python?

    Use the logging package instead of print. Create a module-level logger via logging.getLogger(__name__), configure handlers/formatters at program entry (or with dictConfig). Log at appropriate levels (DEBUG→CRITICAL) and avoid reconfiguring libraries.

39. ## How do you serialize data in Python and when to avoid pickle?

    Use json for interoperable, text-based data; pickle for Python-specific object graphs (not secure against untrusted input). Avoid unpickling data from untrusted sources. For speed, consider orjson/ujson; for schemas, consider pydantic/dataclasses-json.

40. ## How do you handle dates, times, and timezones correctly?

    Use datetime with timezone-aware objects. Store in UTC, display in local time using zoneinfo (Py3.9+) or pytz. Beware naive datetimes; convert explicitly. Use isoformat() for serialization and timedelta for arithmetic.

41. ## What are common regex pitfalls and tips in Python?

    Compile patterns with re.compile for reuse, use raw strings (r"\d+"), understand greedy vs. lazy quantifiers, and anchor with ^/$. Prefer re.escape for literal pieces and named groups for readability.

42. ## How do you build command-line interfaces (CLIs) in Python?

    Use argparse from the standard library for robust parsing, subcommands, and help. Higher-level libraries like Click or Typer simplify ergonomics and typing. Always provide helpful --help and sensible exit codes.

43. ## How do you package a Python project for distribution?

    Use a pyproject.toml (PEP 517/518) to declare build system and metadata. Build wheels (.whl) with tools like build or setuptools. Publish to PyPI via twine. Prefer setup.cfg/pyproject.toml for declarative configs and create editable installs for dev (pip install -e .).

44. ## How do iterators, generators, and itertools help with large data?

    They enable lazy, streaming pipelines that avoid loading everything into memory. Combine itertools tools like islice, chain, groupby, and accumulate to process data efficiently and readably.

45. ## What useful types are in collections?

deque (fast append/pop from both ends), Counter (multiset counting), defaultdict (auto-init values), namedtuple/dataclass (lightweight records), and OrderedDict (rarely needed since Py3.7+ dicts preserve insertion order).

46. What are Abstract Base Classes (ABCs) and when to use them?

ABCs in abc define required methods/properties for a type contract (e.g., collections.abc.Mapping). Subclasses must implement abstract methods. They improve API clarity and enable isinstance checks against behaviors rather than concrete classes.

47. How can you optimize memory usage in Python objects?

Use __slots__ to avoid per-instance __dict__ when attribute names are fixed; prefer arrays/array/memoryview for large numeric buffers; reuse objects where safe; choose tuples over lists for fixed data; stream data instead of materializing lists.

48. How do you speed up CPU-bound Python code?

Profile first. Options include vectorizing with NumPy, using Cython/Numba, writing C/C++/Rust extensions, or moving hot paths to C-accelerated libraries. Multiprocessing can parallelize across cores; avoid excessive IPC overhead.

49. What's the difference between WSGI and ASGI?

WSGI is a synchronous interface used by frameworks like Django/Flask for traditional web apps. ASGI supports async and long-lived connections (WebSockets) and is used by FastAPI/Starlette/Django-ASGI. Choose ASGI for high-concurrency I/O or real-time features.

50. What are key security best practices in Python?

Never eval untrusted input; validate and sanitize data; use parameterized SQL queries; store secrets with environment variables or secret managers (use secrets for cryptographic randomness, not random); keep dependencies updated and pin versions; use safe loaders (e.g., yaml.safe_load).

51. What is the Global Interpreter Lock (GIL) and how does it affect threading?

The GIL allows only one Python bytecode-executing thread at a time in CPython. I/O-bound threads can still overlap, but CPU-bound code won't scale across cores with threads. Workarounds: multiprocessing, native extensions (NumPy/Cython), or async I/O.

52. When should you use threading vs multiprocessing?

Use threading for I/O-bound concurrency (network/file waits). Use multiprocessing for CPU-bound parallelism across cores. Processes avoid the GIL but have higher startup and IPC overhead.

53. Explain asyncio basics: coroutine, event loop, task, and await.

Coroutines (async def) declare async operations. The event loop schedules them. A Task wraps a coroutine so it runs concurrently. await yields control until an awaited awaitable completes. Use asyncio.run(main()) as the entry point.

54. What's the difference between await, asyncio.create_task(), and asyncio.gather()?

await runs an awaitable and pauses the caller. create_task() schedules a coroutine to run concurrently and returns a Task (fire-and-await later). gather() awaits multiple awaitables concurrently and aggregates their results/errors.

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

A shallow copy (copy.copy) copies the container but references the same nested objects. A deep copy (copy.deepcopy) recursively copies nested objects. Classes can customize via __copy__ and __deepcopy__.

56. How does Python's garbage collection work?

CPython uses reference counting for immediate reclamation plus a cyclic garbage collector to detect reference cycles. gc module can tune/inspect it. Cycles with __del__ methods can be problematic and may need weakrefs or explicit cleanup.

57. What are weak references and when are they useful?

weakref lets you reference an object without increasing its ref count, allowing it to be collected. Useful for caches and mappings that shouldn't keep objects alive (WeakKeyDictionary, WeakValueDictionary).

58. What advanced features do dataclasses provide?

field(default_factory=...), control via init, repr, compare, immutability with frozen=True, ordering with order=True, performance with slots=True, keyword-only with kw_only=True, and post-init hooks via __post_init__.

59. How do advanced type hints like Protocol, TypedDict, and Literal help?

Protocol enables structural subtyping; TypedDict types precise dict shapes; Literal restricts to specific values. Combined with generics and Final/Annotated, they improve static checking and documentation.

60. What is structural pattern matching in Python?

Using match/case (PEP 634), you can match on structure and types, destructure variables, and use guards. Example: match msg: case {"type": "point", "x": x, "y": y} ....

61. ## How do you write a proper decorator and preserve function metadata?

    Wrap the target function inside another function and return it; use functools.wraps(func) on the wrapper to copy __name__, __doc__, and __module__. For parameterized decorators, add an outer factory function.

62. ## How do you cache function results in Python?

    Use functools.lru_cache(maxsize=...) or @cache (unbounded, Py3.9+). It memoizes by arguments. Invalidate by calling cache_clear(). For TTLs or custom keys, use external caches or write a custom decorator.

63. ## What are other useful functions in functools?

    partial prebinds arguments; singledispatch enables generic functions by type; reduce folds iterables; cmp_to_key adapts old compare functions for sort; cached_property lazily computes and caches attributes.

64. ## What problem do contextvars solve?

    contextvars provide context-local state that propagates correctly in async code (unlike thread-locals). Useful for request IDs, auth context, or tracing across awaiting boundaries.

65. ## What are best practices for running external commands?

    Prefer subprocess.run([...], check=True) with a list of args, set timeout=..., capture output with text=True, capture_output=True, avoid shell=True unless necessary (then sanitize), and close resources promptly.

66. ## How should you manage configuration and environment variables?

    Read via os.getenv/os.environ, supply sensible defaults, and validate. Use .env loaders or libraries like Pydantic's BaseSettings. Never hardcode secrets; rely on secret managers or environment injection.

67. ## How do you expose a Python CLI as an installable command?

    Declare project.scripts/entry_points.console_scripts in pyproject.toml/setup.cfg. After install, a console wrapper is generated (usable with pipx for isolated app installs).

68. ## What are virtual environments and how do you pin dependencies?

    Create envs with python -m venv (or uv, conda, poetry). Pin exact versions in a lock file (requirements.txt via pip-tools, poetry.lock). Use version ranges carefully (~=, ==, >=,<).

69. ## What are key pytest features to learn first?

    Fixtures (with scopes), @pytest.mark.parametrize, pytest.raises, tmp_path for temp files, and monkeypatch for environment/attribute overrides. Tests are simple functions with plain asserts.

70. ## How does mocking work in Python testing?

    Use unittest.mock or pytest's wrapper. patch replaces attributes within a context; Mock/MagicMock create fakes; spec/autospec=True enforce real API; inspect calls via assert_called_with.

71. ## How do you profile performance and memory usage?

    Use timeit for micro-benchmarks, cProfile/pstats or pyinstrument for runtime profiles, line_profiler for per-line timing, and tracemalloc for memory allocations and leak detection.

72. ## Why are NumPy arrays faster than Python lists for numeric work?

    NumPy stores homogeneous data in contiguous memory and executes vectorized operations in optimized C code, reducing Python loop overhead and enabling SIMD/BLAS acceleration.

73. ## How can Python interface with C/C++ for speed?

    Options include C extensions via CPython API, Cython, ctypes, cffi, or C++ with pybind11. Choose based on complexity: wrappers (ctypes/cffi) vs compiled modules (Cython/pybind11) for hot paths.

74. ## What should you know about Unicode and i18n in Python?

    Strings are Unicode; encode/decode explicitly (UTF-8). Normalize when comparing (unicodedata.normalize). For localization use gettext and locale/babel for formatting numbers/dates.

75. ## What are common Python idioms and anti-patterns?

    Prefer EAFP over LBYL (try/except rather than pre-checks), use context managers for resources, avoid catching bare Exception, don't use mutable default args, and favor comprehensions/generators over manual loops when clearer.

76. ## What is the Python data model and why is it important?

    The data model is the set of protocols (dunder methods) that define how objects behave—construction, representation, iteration, arithmetic, comparison, attribute access, context management, etc. By implementing methods like len, iter, getitem, eq, and enter/exit,

your objects integrate naturally with the language and standard library.

77. ### What's the difference between str and repr?

    repr should return an unambiguous representation primarily for developers (ideally valid Python to recreate the object). str is a readable display string for end users. If only one is defined, Python falls back to repr.

78. ### How do you correctly implement equality and hashing for custom classes?

    Define eq to compare logical identity and, if instances are hashable, also define hash consistently (equal objects must have equal hashes). For immutable classes, use @dataclass(frozen=True) or functools.total_ordering to reduce boilerplate.

79. ### What is a descriptor in Python?

    A descriptor is an object with get, set, or delete that manages attribute access. It powers properties, functions (as bound methods), and many frameworks (e.g., Django fields). Use descriptors to centralize validation, caching, or computed attributes.

80. ### What are slots and when should you use them?

    slots restricts instance attributes to a fixed set and stores them in a compact C array rather than a dict, reducing memory and speeding attribute access. Downsides: no dict per instance, limited flexibility, inherits intricacies, and weakref support requires weakref in slots.

81. ### What are Abstract Base Classes (ABCs)?

    ABCs define interfaces with abstract methods via abc.ABC and @abstractmethod. They allow isinstance checks by behavior (e.g., collections.abc.Sequence). You can register virtual subclasses with .register() without inheritance.

82. ### When would you use a metaclass?

    Metaclasses customize class creation (e.g., auto-register subclasses, enforce constraints, create fields from annotations). They're powerful but complex; prefer simpler tools first (dataclasses, decorators, descriptors) before metaclasses.

83. ### How do you create a context manager?

    Implement enter and exit on a class or use @contextlib.contextmanager with a generator that yields exactly once. Context managers guarantee cleanup (files, locks, DB connections) even when exceptions occur.

84. ### What are best practices for file I/O in Python?

    Use context managers: with open(path, mode, encoding='utf-8', newline='') as f: ... Prefer pathlib.Path for paths and portability; handle large files with iterators/chunks; explicitly set encoding and newline for text files.

85. ### How do you configure logging for an application?

    Use the logging package with levels (DEBUG...CRITICAL), formatters, and handlers (Stream/File/Rotating). Prefer dictConfig or basicConfig at app entry; avoid print for prod; include contextual data with LoggerAdapter or extra=.

86. ### Compare JSON, Pickle, and Marshal for serialization.

    JSON is text, interoperable, and safe; use it for APIs/configs. Pickle is Python-specific and can execute arbitrary code on load—never unpickle untrusted data. marshal is internal to Python and version-dependent; avoid for app data.

87. ### How do you work with time zones in Python?

    Use timezone-aware datetimes. Since 3.9, zoneinfo provides IANA time zones: from zoneinfo import ZoneInfo; dt = datetime.now(ZoneInfo('UTC')). Convert with .astimezone(). Avoid naive arithmetic across DST boundaries.

88. ### When should you use Decimal or Fraction instead of float?

    Use Decimal for exact base-10 arithmetic (money/taxes) and configurable precision; use Fraction for exact rational math; floats are fast but binary and can accumulate rounding error.

89. ### What are some useful itertools tools?

    count, cycle, repeat, accumulate, chain, islice, groupby, product, permutations, combinations, and tee. They enable lazy, memory-efficient pipelines for combinatorics and streaming transforms.

90. ### Which specialized containers are in collections and when to use them?

    deque for fast append/pop from both ends; Counter for tallies; defaultdict for auto-initialized dicts; ChainMap for layered configs; namedtuple/dataclass for lightweight records; OrderedDict mainly for legacy (<3.7).

91. ### How do generator methods send, throw, and close work?

    gen.send(value) resumes a generator injecting a value to the last yield; gen.throw(exc) raises inside the generator; gen.close() raises GeneratorExit to end it. 'yield from subgen' delegates these operations to a subgenerator.

92. ### What's the difference between iterables, iterators, and generators?

An iterable implements iter returning an iterator. An iterator implements iter and next. A generator is an iterator created by a generator function (yield) or expression; it auto-implements the protocol.

93. ### How can you reduce memory usage in Python programs?

    Use generators instead of lists, slots for many small objects, array/bytes/memoryview for numeric/byte data, share interned strings, avoid retaining references (clear caches), and profile with tracemalloc.

94. ### How do you package a Python project for distribution?

    Use pyproject.toml (PEP 517/518) to declare build system and metadata, then build wheels/sdists with tools like build or hatch. Publish to PyPI with twine. Include dependencies, license, classifiers, and entry points.

95. ### What are ways to manage and pin dependencies?

    Use virtual environments, pin versions in requirements.txt or a lockfile. Tools: pip-tools (compile), Poetry or Hatch (lock + environment). Prefer compatible pinning (~=) for libs; exact pins for apps.

96. ### Explain Python's import system and package layout best practices.

    A module is a .py file; a package is a directory with init.py (namespace packages can omit it). Prefer absolute imports; use relative only within a package. Avoid modifying sys.path; use proper packaging instead.

97. ### How do you build a command-line interface (CLI) in Python?

    Use argparse for standard library parsing, or libraries like click/typer for ergonomics and rich help. Expose a console script via entry_points (or project.scripts) so users can run it as a command after install.

98. ### Compare Flask and FastAPI for building web APIs.

    Flask is a minimal WSGI microframework with vast ecosystem. FastAPI is modern ASGI-first, type-hint driven, with automatic data validation and docs via Pydantic and OpenAPI, and excellent async support. Choose based on sync/async needs and ergonomics.

99. ### What should you know about SQLAlchemy ORM sessions?

    A Session manages a unit of work and identity map. Objects are transient → pending → persistent → detached. Use scoped_session or sessionmaker; commit flushes changes; close or use context managers to release connections.

100. ### What are key security best practices for Python applications?

     Avoid eval/exec on untrusted input; never unpickle untrusted data; prefer subprocess without shell=True and validate inputs; store secrets in env/secret managers; keep dependencies updated; validate/escape user input; enable logging and limit exception details in production.

101. ### What is a typing.Protocol and when would you use it?

     Protocol defines a structural (duck-typed) interface. If a class implements the required methods/attributes, it satisfies the Protocol without explicit inheritance. Use it to type functions that accept any object with a given interface (e.g., 'file-like' objects with .read).

102. ### Explain TypeVar, Generic, and bounded TypeVars.

     TypeVar creates a type parameter; Generic marks a class/function as parameterized. Bounded TypeVars (TypeVar('T', bound=Base)) restrict a parameter to a subtype of Base. They enable reusable, type-safe containers and APIs.

103. ### What are ParamSpec and Concatenate in typing?

     ParamSpec captures a callable's parameter types to forward them (e.g., decorators preserving signatures). Concatenate lets you prepend/append parameters when wrapping callables. They improve type accuracy of higher-order functions.

104. ### Compare TypedDict, dataclass, and plain dict for structured data.

     TypedDict adds static types to dict keys but remains a runtime dict. dataclass creates lightweight classes with fields, defaults, and methods (hash/eq). Plain dict is flexible but untyped. Choose based on need for validation, methods, and performance.

105. ### What are Literal, Final, and Annotated used for?

     Literal restricts a value to specific constants; Final marks names or attributes that shouldn't be reassigned/overridden; Annotated attaches metadata to a type (e.g., validation hints) consumed by frameworks and tools.

106. ### What advanced features do dataclasses provide?

     Useful options include frozen (immutability), slots (memory savings), kw_only (keyword-only fields), default_factory, init=False, order=True, field metadata, and hooks like post_init for validation or derived fields.

107. ### How does structural pattern matching work and when is it appropriate?

     match/case can match literals, sequences, mappings, classes (via match_args), and guards (if conditions). Use it to replace nested if/elif for parsing tokens, ASTs, or protocol messages. Keep patterns explicit and avoid overly clever matches.

108. ## How do you correctly cancel asyncio tasks?

Store the Task, call task.cancel(), then await it in a try/except asyncio.CancelledError block to let it clean up. Use asyncio.timeout() or shield() to control cancellation boundaries; always design coroutines to handle cancellation at awaits.

109. ## What is flow control/backpressure in asyncio streams?

Backpressure prevents fast producers from overwhelming slow consumers. In streams, await writer.drain() yields when the buffer is full; in higher-level code use semaphores/queues to limit concurrency and enforce bounded buffering.

110. ## When should you choose threads, multiprocessing, or asyncio?

Threads are great for I/O-bound work with blocking libraries. Multiprocessing sidesteps the GIL for CPU-bound tasks at IPC cost. asyncio is best for high-concurrency I/O with async-aware libraries. Often combine: processes for CPU, async for I/O.

111. ## What are contextvars and how do they differ from threading.local?

contextvars store context-local state that propagates correctly across async task switches, unlike threading.local which is thread-local only. Use contextvars for request-scoped data in async apps (e.g., correlation IDs).

112. ## Explain memoryview and zero-copy slicing.

memoryview exposes the buffer protocol of bytes/bytearray/array without copying. Slicing a memoryview returns a view, not a new allocation. Use it for efficient binary parsing and interop with I/O and NumPy.

113. ## How do you profile performance and memory in Python?

Use timeit for micro-benchmarks; cProfile or profile + pstats for CPU hot spots; line_profiler for per-line timing; tracemalloc for memory allocation sources; perf_counter for wall time. Always measure with realistic inputs.

114. ## What is the garbage collector’s role with reference cycles?

CPython uses reference counting plus a cyclic GC to reclaim objects in reference cycles. Cycles with objects defining del can be problematic; prefer weakref or context managers to avoid cycles and ensure prompt cleanup.

115. ## How do you efficiently build large strings?

Accumulate parts in a list and ''.join(parts), or use io.StringIO for repeated appends. Avoid repeated s += piece in tight loops to reduce quadratic copying.

116. ## What is all and how does it affect imports?

all is a list of public names exported by a module when using 'from module import *'. It also serves as documentation of the intended public API. It doesn’t affect direct attribute access via module.name.

117. ## How do importlib.resources help with package data files?

importlib.resources provides APIs (files, open_text, as_file) to access data bundled in packages in a loader-agnostic way. Use it instead of file hacks so it works with zipped wheels and different importers.

118. ## What are weak references and typical use cases?

weakref.ref lets you reference objects without increasing their refcount, allowing GC when no strong refs remain. Use for caches, observer lists, or mapping large objects without preventing collection.

119. ## How do functools.lru_cache and cached_property differ?

lru_cache memoizes function calls by arguments with a size-limited cache. cached_property computes once per instance and stores the value on the instance. Use lru_cache for pure functions; cached_property for expensive attribute computation.

120. ## What is singledispatch and when should you use it?

functools.singledispatch enables function overloading based on the first argument’s type. Register implementations for different types to keep polymorphic logic organized without modifying classes.

121. ## Which standard modules help with algorithmic efficiency?

bisect for binary search and sorted insertion; heapq for priority queues; collections.deque for O(1) appends/pops; array for compact numeric storage; itertools for lazy pipelines; statistics for basic metrics.

122. ## How do you avoid circular import problems?

Prefer module-level design with clear dependency direction, move shared types/interfaces to a common module, use local imports inside functions when necessary, and refactor tightly coupled modules into packages.

123. ## What are best practices for resilient network I/O in Python?

Set timeouts on sockets/HTTP, implement retries with jittered exponential backoff, handle specific exceptions, bound concurrency (semaphores/connection pools), and add circuit-breaking/fail-fast behavior for dependencies.

124. ## What are common regex performance pitfalls and mitigations?

Catastrophic backtracking arises from nested quantifiers on ambiguous patterns. Use non-greedy quantifiers, atomic alternatives (more specific patterns), anchors/boundaries, pre-compile with re.compile, and consider re2-like libs if needed.

125. ## How do you safely evaluate or parse untrusted input?

Never use eval/exec on untrusted text. For Python literals, use ast.literal_eval. For expressions, build a small DSL or parse explicitly. Isolate untrusted code in a separate process/container with strict resource limits.

126. ## What is pyproject.toml and why is it important?

pyproject.toml centralizes build and tooling config. It declares the build backend (e.g., setuptools, hatchling, poetry), project metadata, and tool sections (black, isort, mypy). It’s the modern, PEP-517/518 compliant way to package Python projects.

127. ## How do console scripts (entry points) work?

Packaging tools expose functions as CLI commands via entry points (e.g., console_scripts in setuptools). On install, a launcher script is generated that imports your module and calls the target function.

128. ## Compare venv, virtualenv, pipx, and conda.

venv (stdlib) creates lightweight virtual environments. virtualenv is similar with extras and speedups. pipx installs and isolates CLI apps in their own envs. conda is a cross-language env+package manager (binaries) popular in data science.

129. ## How does Python find and load modules?

The import system consults sys.meta_path finders to locate a module, then a loader to create it. sys.path (directories, wheels, zips) drives where searching happens. Packages may have init.py (regular) or be namespace packages.

130. ## What are namespace packages?

Namespace packages (PEP 420) are packages without init.py that can span multiple directories/distributions under the same top-level name. They simplify large, split codebases and plugin ecosystems.

131. ## Why use python -m package.module and what is __main__.py?

python -m executes a module as a script using its package context (reliable relative imports). A package with main.py can be run as python -m package or zipped and executed as an application.

132. ## What does __slots__ do and when should you use it?

__slots__ removes per-instance dicts, saving memory and speeding attribute access. Trade-offs: no new attributes unless in slots, limited multiple inheritance, and some tooling friction. Great for many small objects.

133. ## Explain the descriptor protocol.

Descriptors define any of get, set, delete on a class attribute to control attribute access (e.g., property, functions as bound methods). Useful for validation, computed attributes, and ORMs.

134. ## Difference between getattr and getattribute?

getattr(name) is called only if normal attribute lookup fails. getattribute(name) intercepts every attribute access—must delegate to super().getattribute to avoid infinite recursion.

135. ## How does @property work and when to use setters/deleters?

@property creates managed attributes via descriptor methods. Use when you want attribute syntax but computed/validated behavior; add @x.setter/@x.deleter for controlled mutation without breaking API.

136. ## How do eq and hash relate, and what pitfalls exist?

Equal objects must have the same hash. If you define eq but not hash, instances may become unhashable. For mutable objects, either avoid hashing or base equality on immutable identity.

137. ## How do you implement ordering on custom classes?

Implement rich comparisons (lt/le/gt/ge). You can define one (e.g., lt) plus equality and use functools.total_ordering to auto-fill the rest, given a consistent key.

138. ## What advanced tools exist for context managers?

contextlib.contextmanager for generator-based managers, ExitStack to compose multiple managers dynamically, closing for objects with close(), and suppress to ignore specified exceptions.

139. ## Why prefer pathlib over os.path?

pathlib provides object-oriented, cross-platform paths with methods for I/O, globbing, reading/writing text/binary, and easy joining (path / 'file.txt'). It’s clearer and less error-prone.

140. What are logging best practices?

Use the logging package with levels, handlers, and formatters; avoid print for diagnostics. Structure logs, add contextual info, configure via dictConfig, and don’t log secrets or high-volume data at INFO.

141. What are pycache and .pyc files?

CPython caches compiled bytecode in pycache as .pyc files (tagged by version/ABI) to speed startup. They’re auto-managed; invalidated when source changes or interpreter version changes.

142. What are Exception Groups and except*?

Exception groups (BaseExceptionGroup) represent multiple errors at once (e.g., concurrent tasks). except* handles subsets by type. Useful in asyncio/parallel code to report partial failures cleanly.

143. What is asyncio.TaskGroup and why use it?

TaskGroup runs child tasks as a unit: start several coroutines, await the group, and propagate failures predictably. It simplifies structured concurrency compared to manual task management.

144. How do async context managers and async iterators work?

Use async with to manage async resources (defines aenter/aexit) and async for to consume async iterables (defines aiter/anext). They integrate await points for non-blocking I/O.

145. What are common multiprocessing pitfalls on different OSes?

On Windows/macOS ‘spawn’ is default—guard code with if __name__ == "__main__" and ensure targets are pickleable. Avoid global state; prefer queues/pipes for IPC; be mindful of copy-on-write differences vs Linux ‘fork’.

146. What should you know about pickling?

Pickle is Python-specific and unsafe for untrusted data. Classes can customize via getstate/setstate or reducers. Prefer interoperable formats (JSON/MsgPack) for external data and version your schemas.

147. When to use secrets vs random?

Use secrets for security-sensitive tokens, passwords, and salts (cryptographic strength). random is for simulations or non-security tasks. Don’t derive secrets from random.random or predictable seeds.

148. What does it mean for an object to be hashable?

Hashable objects have a stable hash and equality that doesn’t change over their lifetime; they can be dict keys or set members. Mutable types generally shouldn’t be hashable to avoid broken lookups.

149. How do you speed up CPU-bound Python code?

Use optimized libraries (NumPy), vectorize, move hotspots to C/Cython, JIT with Numba (where applicable), or use multiprocessing for parallelism. Profile first; avoid premature optimization.

150. What are testing best practices in Python?

Use pytest for concise tests, fixtures for setup, parametrization for cases, monkeypatch for isolation, and Hypothesis for property-based testing. Keep tests deterministic, fast, and independent.

151. What is structural pattern matching in Python?

Structural pattern matching (PEP 634–636) lets you match on a value’s shape using match/case. It supports literals, sequences, mappings, class patterns, and wildcards. Example: match obj: case {"id": int(x), "name": str(n)}: ....

152. How do you match sequences, mappings, and classes with match?

Sequences: case [a, b, *rest]. Mappings: case {"id": id, **rest}. Classes: define __match_args__ or use keyword attributes: case Point(x=x, y=y). Guards add conditions: case [x, y] if x < y:.

153. What are as-patterns and guards in pattern matching?

as captures a matched value: case [x, y] as pair: binds pair to the list. Guards are if clauses that further constrain a match: case int(x) if x % 2 == 0:.

154. What problems do @dataclasses solve and which options matter most?

@dataclass generates boilerplate (__init__, __repr__, comparisons). Key options: frozen=True (immutability), order=True (ordering methods), slots=True (memory/perf), and kw_only=True (keyword-only fields).

155. When and how do you use __post_init__ in a dataclass?

Use __post_init__ for validation or derived fields after auto __init__ runs. Example: def __post_init__(self): assert self.count >= 0; self.slug = slugify(self.name).

156. ## Why prefer default_factory over a mutable default in dataclasses?

Mutable defaults are shared between instances. Use field(default_factory=list) (or dict/set) to get a fresh object per instance and avoid surprising mutations.

157. ## Compare dataclass, NamedTuple, and TypedDict.

dataclass: regular (mutable or frozen) classes with methods. NamedTuple: immutable, tuple-like, lightweight with fixed fields. TypedDict: typing-only dict schema for dynamic dicts; runtime is still a dict.

158. ## What is the buffer protocol and why use memoryview?

The buffer protocol exposes raw bytes of objects (bytes, bytearray, NumPy arrays). memoryview provides zero-copy slicing/transformations on binary data, reducing allocations for I/O and parsing.

159. ## Bytes vs bytearray vs memoryview—when to choose each?

bytes = immutable binary data (hashable). bytearray = mutable for in-place edits. memoryview = a zero-copy window over another buffer; ideal for slicing and I/O without copying.

160. ## When should you use decimal.Decimal instead of float?

Use Decimal for exact base-10 arithmetic (money, invoices) and configurable precision/rounding. Floats are binary and can’t represent many decimal fractions exactly (0.1 issues).

161. ## How do you handle time zones with zoneinfo?

Use aware datetimes with zoneinfo.ZoneInfo("America/New_York"). Convert via .astimezone(). Store in UTC, display in local time, and avoid naive datetimes for cross-TZ data.

162. ## What is importlib.resources used for?

It loads package data files safely from wheels/zips. Use files(pkg)/"data.json" and read_text()/read_bytes() or as_file() to get a real path when needed.

163. ## How do you safely run external commands with subprocess?

Prefer subprocess.run(["prog", "arg"]) with a list, not a shell string. Avoid shell=True unless necessary. Use check=True, capture_output=True, and timeouts for robustness.

164. ## How do you prevent SQL injection with sqlite3 and improve results usability?

Use parameterized queries: cur.execute("SELECT ... WHERE id=?", (id_,)). Set conn.row_factory = sqlite3.Row for dict-like rows (row["col"]).

165. ## What does functools.lru_cache do and what are caveats?

It memoizes function results by arguments: @lru_cache(maxsize=1024, typed=True). Beware of mutable args (must be hashable), memory growth, and cache invalidation (cache_clear).

166. ## How do you profile memory usage with tracemalloc?

Start early: tracemalloc.start(). Take snapshots and compare: s1=tracemalloc.take_snapshot(); ...; s2=...; s2.compare_to(s1, "lineno") to find top allocators.

167. ## When to use ThreadPoolExecutor vs ProcessPoolExecutor?

Threads for I/O-bound tasks (overlap waits; GIL isn’t a big issue). Processes for CPU-bound workloads (bypass GIL). Consider startup/IPC overhead and pick chunk sizes accordingly.

168. ## What is contextvars and how does it differ from threading.local?

contextvars stores context that propagates correctly in async tasks; it’s task-local. threading.local is thread-local and doesn’t flow across await boundaries.

169. ## How do you stream large files efficiently in Python?

Use buffered iteration and chunked reads: for chunk in iter(lambda: f.read(1024*1024), b""):. Avoid loading entire files into memory; write chunks to the destination.

170. ## How can you speed up and control regex behavior?

Precompile patterns: pat = re.compile(..., flags). Use re.VERBOSE for clarity. In recent Python, atomic grouping (?>(...)) and possessive quantifiers (++, *+) can reduce catastrophic backtracking.

171. ## How do you build a CLI with subcommands using argparse?

Create a parser, then subparsers = parser.add_subparsers(dest="cmd"); add each subparser with its own options. Dispatch by args.cmd or set a default function with set_defaults(func=...).

172. ## What's the difference between shallow and deep copy?

Shallow copy (copy.copy) copies the outer container but references inner objects. Deep copy (copy.deepcopy) recursively copies contents. Customize via __copy__/__deepcopy__ when needed.

173. ## How can you implement a plugin architecture in pure Python?

Discover entry points via importlib.metadata.entry_points() (packaged plugins) or import modules from a namespace package. Load callables dynamically and register them in a central registry.

174. ## What are weakref and WeakValueDictionary useful for?

They reference objects without increasing reference counts. When objects are GC'd, weak refs clear automatically—useful for caches and avoiding memory leaks due to reference cycles.

175. ## How do you implement a custom context manager?

Define __enter__/__exit__ on a class, or use @contextmanager with a generator. Example: @contextmanager def timer(): start=time.perf_counter(); try: yield; finally: print(time.perf_counter()-start).

176. ## What are PEP 517/518 and pyproject.toml?

PEP 517/518 define a standardized way to declare and use build backends for packaging. pyproject.toml is the canonical file where you specify the build system (e.g., build-system.requires = ["setuptools", "wheel"], build-backend = "setuptools.build_meta") and often tool configs (black, isort, mypy).

177. ## What is a wheel and what does manylinux mean?

A wheel (.whl) is a built, ready-to-install package format that avoids compiling at install time. "manylinux" wheels are Linux binary wheels built against a stable ABI standard so they run on many distros without recompilation.

178. ## How do you make and turn on a virtual environment?

Create it: python -m venv .venv
Turn it on:
- macOS/Linux: source .venv/bin/activate
- Windows (PowerShell): .\.venv\Scripts\Activate.ps1
- Windows (CMD): .\.venv\Scripts\activate.bat
This keeps your project's packages separate from the system Python.

179. ## Difference between a module, a package, and a namespace package?

Module: a single .py file. Package: a directory with __init__.py that groups modules. Namespace package: a package without __init__.py whose parts can be spread across multiple directories/distributions.

180. ## How do you publish a package to PyPI securely?

Build with python -m build (sdist + wheel). Upload with twine upload dist/* using a PyPI API token (store in ~/.pypirc or env var). Never embed credentials in code; sign releases if needed.

181. ## How does CPython manage memory?

CPython uses reference counting for immediate reclamation and a cyclic garbage collector that periodically finds and frees reference cycles across three generations. Small objects come from specialized allocators (obmalloc).

182. ## When are objects deallocated immediately vs. later?

When an object's refcount hits zero, CPython often frees it immediately (e.g., local temporaries). Objects in cycles or with delayed finalization are collected later by the cyclic GC, so __del__ timing isn't guaranteed.

183. ## How does the GIL affect CPU-bound code and how can you work around it?

The GIL allows only one Python bytecode thread to execute at a time, limiting CPU-bound speed-ups with threads. Use multiprocessing, offload hot loops to C/Cython/Numba, vectorize with NumPy, or consider PyPy for some workloads.

184. ## Compare multiprocessing start methods: fork, spawn, forkserver.

fork (Unix): fast, copies the parent memory (COW) but can be unsafe with threads. spawn (default on Windows/macOS): starts fresh interpreter—safer but slower. forkserver (Unix): central server forks new children safely.

185. ## What is copy-on-write and why does it matter for multiprocessing?

With fork, the child initially shares the parent's memory pages. Pages are only copied when written to, saving memory and speeding startup for large data. Mutations break sharing; spawn doesn't benefit from this.

186. ## How are OS signals handled in Python?

Signals are delivered to the main thread. Use signal.signal(SIGINT, handler) or, with asyncio on Unix, loop.add_signal_handler. Windows has limited signal support; KeyboardInterrupt maps to SIGINT (Ctrl-C).

187. What are contextlib.ExitStack and AsyncExitStack for?

They manage a dynamic set of context managers (entered conditionally or in loops). You can push callbacks or context managers and guarantee orderly exit/cleanup even when the set isn't known statically.

188. What are descriptors and where are they used?

Descriptors implement __get__, __set__, __delete__ to control attribute access. They power @property, methods (functions are non-data descriptors), @classmethod, @staticmethod, ORM fields, and validators.

189. What does __slots__ do and when should you use it?

__slots__ fixes the set of attributes and removes per-instance __dict__, reducing memory and attribute lookup overhead. Trade-offs: no new attributes unless in slots, inheritance caveats, and limited compatibility with some tools.

190. Explain the __eq__/__hash__ contract.

If you define __eq__, Python sets __hash__ = None (unhashable) unless you also define a compatible __hash__. Equal objects must have equal hashes. For dataclasses: frozen=True yields a safe hash by default.

191. What are typing.Protocol and structural subtyping?

Protocols define method/attribute requirements that any object can satisfy by shape, not inheritance. With @runtime_checkable, you can use isinstance(x, Proto). Protocols complement or replace ABCs for duck typing.

192. How does variance work in Python typing?

Generics are invariant by default. Some standard types are covariant (e.g., Sequence[T_co]) or contravariant (e.g., callable arguments). You declare variance with TypeVar("T", covariant=True, ...) when creating APIs.

193. How do you narrow types effectively when using type checkers?

Use isinstance checks, typing.Literal for exact values, TypedDict for dict schemas, assert statements, and typing.cast when you must inform the checker of a safe assumption.

194. What are functools.singledispatch and singledispatchmethod?

singledispatch enables function overloading on the first argument's type. Register specialized implementations with @func.register(SomeType). singledispatchmethod brings the same to instance methods.

195. How do you optimize numeric code in Python?

Avoid Python loops; use NumPy vectorization and broadcasting. JIT compile with Numba for nopython kernels, write critical paths in C/Cython, or use PyPy. Minimize Python↔NumPy boundary crossings.

196. Explain NumPy broadcasting rules.

Align shapes from the trailing dimensions; dimensions are compatible if equal or one of them is 1. NumPy virtually stretches size-1 dimensions without copying, enabling element-wise ops across different shapes.

197. What causes SettingWithCopyWarning in pandas and how do you avoid it?

It occurs with chained indexing that may act on a view, not a copy. Use .loc[row_sel, col_sel] = ... in a single step or call .copy() explicitly before modifying.

198. How do you handle large file I/O efficiently?

Stream in chunks, use buffered I/O, and consider mmap for random access. For CSVs, use pandas read_csv(..., chunksize=...). Avoid reading entire files into memory when not necessary.

199. What are logging best practices in Python?

Prefer logging over print. Create module-level loggers (logger = logging.getLogger(__name__)), configure handlers/formatters once (JSON if needed), don't spam root logger, and use appropriate levels with contextual extras.

200. What security practices should you follow in Python projects?

Avoid eval/exec on user data, use secrets for tokens (not random), hash passwords with bcrypt/scrypt/argon2, validate inputs, pin dependencies, scan with Bandit/Safety, and store secrets in env/secret managers.

201. What is structural pattern matching in Python and when should you use it?

Structural pattern matching (PEP 634–636, match/case) lets you branch on the shape of data—sequences, mappings, dataclasses—rather than just values. Use it to replace long if/elif chains, to destructure inputs, and to make intent clear for parsers, protocol handlers, and AST/JSON processing.

202. ## What are ExceptionGroup and except* used for?

ExceptionGroup groups multiple exceptions into one (e.g., from concurrent tasks). except* handles members of a group selectively, allowing partial handling while the rest propagate. This is especially useful with asyncio or task pools where many errors can occur together.

203. ## What is asyncio.TaskGroup and how is it different from creating tasks manually?

TaskGroup (3.11+) provides structured concurrency: tasks started inside the group are tracked as a unit; failures cancel siblings and are reported as an ExceptionGroup. It reduces orphaned tasks and error-handling boilerplate compared to ad-hoc create_task usage.

204. ## What are contextvars and why are they important with asyncio?

contextvars hold context-local state (e.g., request IDs) that propagates across async tasks without relying on thread locals. They enable safe, per-task context (logging correlation IDs, tracing) even when tasks share threads.

205. ## Explain key options of dataclasses.field like default_factory, repr, and compare.

default_factory supplies a callable to create a fresh default (e.g., list). repr=False hides the field from __repr__. compare=False excludes it from __eq__/ordering. Together they control defaults and how instances display/compare.

206. ## What do slots=True and kw_only=True do in @dataclass?

slots=True creates __slots__, removing the per-instance __dict__ to save memory and speed attribute access. kw_only=True makes all fields keyword-only in the generated __init__, improving call clarity and preventing arg-mismatch bugs.

207. ## Difference between __new__ and __init__?

__new__(cls, ...) allocates and returns the new instance (used for immutables like int, str, or customizing creation). __init__(self, ...) initializes an already-created instance. If __new__ returns an instance of another class, __init__ will not run.

208. ## How can you define immutable data structures in Python?

Use @dataclass(frozen=True) for dataclasses, typing.NamedTuple or collections.namedtuple for tuple-like records, or implement __setattr__ to forbid mutation. For hashability, ensure fields are themselves immutable.

209. ## How do you implement a custom iterator or generator efficiently?

Implement __iter__ returning self and __next__ raising StopIteration for iterators. Prefer generators (yield) for simplicity. Use yield from to delegate to sub-iterables, which preserves exceptions and reduces boilerplate.

210. ## What is yield from and what problems does it solve?

yield from subiter delegates iteration to subiter, forwarding values, send(), throw(), and close() automatically. It simplifies generator composition (pipelines, coroutines) and returns a final value via StopIteration.value.

211. ## How do you create custom context managers?

Implement __enter__/__exit__ methods on a class or use contextlib.contextmanager to turn a generator into a context manager. Ensure cleanup occurs in __exit__/finally, even when exceptions are raised.

212. ## What are weakrefs and when are they useful?

weakref references an object without increasing its reference count, allowing it to be garbage-collected. Use for caches, memoization, or observer lists where you don’t want to keep objects alive accidentally.

213. ## How do you profile CPU performance in Python?

Use cProfile/pstats for function-level hotspots, time.perf_counter() for timing blocks, and tools like line_profiler or py-spy for deeper insight. Always run representative workloads and avoid profiling with debug builds.

214. ## How do you track memory usage and leaks?

tracemalloc tracks allocations and shows top consumers and diffs between snapshots. Combine with careful object-lifetime analysis (e.g., reference cycles, global caches) and tools like objgraph to find leaks.

215. ## How can you speed up imports or delay heavy dependencies?

Use local imports inside functions, gate imports behind feature flags, or import lazily with importlib. Guard typing-only imports with if TYPE_CHECKING: to avoid runtime cost while keeping precise type hints.

216. ## What is tomllib and when would you use it?

tomllib (3.11+) is a stdlib TOML reader. Use it to parse configuration files (like pyproject.toml) without third-party dependencies. For writing TOML, use an external library.

217. ## How do you safely access data files bundled with a package?

Use importlib.resources (e.g., files(), as_file(), open_text()/open_binary()) to read resources in a way that works both from source trees and installed wheels/zip imports.

218. What's the recommended way to build command-line interfaces in the stdlib?

Use argparse for parsing flags/args, subcommands, and help messages. Combine with dataclasses for config objects and logging for verbosity control (-v).

219. Why prefer pathlib over os.path?

pathlib.Path provides object-oriented paths, operator overloading (/), better readability, and rich methods (glob, read_text, rename). It handles different OS separators transparently.

220. Compare functools.cache, lru_cache, and cached_property.

@cache is an unbounded memoizer for pure functions. @lru_cache(maxsize=N) bounds memory and supports .cache_clear(). @cached_property caches a computed attribute per instance after first access.

221. How do you share memory efficiently between processes?

Use multiprocessing.shared_memory (3.8+) for zero-copy sharing of raw buffers/NumPy arrays, or multiprocessing.Array/Value for simple C types. Managers/proxies are simpler but add IPC overhead.

222. How do positional-only and keyword-only parameters work and why use them?

Use / to mark parameters positional-only and * to mark parameters keyword-only: def f(a, /, b, *, c). This stabilizes APIs (call sites can't rely on names), clarifies intent, and avoids accidental arg reordering.

223. What is typing.Annotated and how is it used?

Annotated[T, meta...] attaches metadata to a type—constraints, descriptions—used by tools (validators, DI, docs) while remaining T at runtime. Retrieve with typing.get_type_hints(include_extras=True).

224. What is typing.Self and where is it helpful?

Self expresses that a method returns an instance of the same class (including subclasses). It improves typing for fluent builders and classmethod constructors without repeating the class name.

225. Why is using pickle with untrusted data dangerous and what should you use instead?

pickle can execute arbitrary code during deserialization, making it unsafe for untrusted inputs. Prefer safe formats like JSON/TOML/YAML (with a safe loader), or domain-specific schemas (e.g., pydantic, dataclasses + json).

226. What's the difference between WSGI and ASGI?

WSGI is a synchronous interface (Flask/Django classic) for HTTP requests—one request per worker thread/process. ASGI is asynchronous and supports long-lived connections (websockets), background tasks, and concurrency via event loops (Starlette/FastAPI/Django async). Choose ASGI for real-time/IO-heavy workloads.

227. When would you choose FastAPI vs Flask vs Django?

FastAPI: modern async APIs with automatic validation/docs via Pydantic; great performance. Flask: minimal, flexible micro-framework; add only what you need. Django: batteries-included (ORM, admin, auth) for full-stack apps and rapid CRUD.

228. How do you structure a large Python project?

Use a src-layout (src/yourpkg), pyproject.toml for builds/deps, tests/ for pytest, and separate runtime config from code. Keep clear package boundaries, type hints, pre-commit hooks (ruff/black/isort), and CI to run lint+tests.

229. What are common environment management options?

venv/virtualenv for per-project isolation; pipx for global CLI tools; Conda/Mamba for Python+native libs stacks. Pin versions (requirements.txt or lockfiles) and avoid system Python for apps.

230. How do Poetry and pip-tools differ?

Poetry manages pyproject deps, locking, build/publish in one tool. pip-tools (pip-compile) turns declarative requirements.in into pinned requirements.txt. Poetry is holistic; pip-tools is lightweight and pip-native.

231. Explain wheels, sdists, and manylinux.

sdist is source; wheel is a built package for fast installs. manylinux wheels bundle compiled native code compatible with many Linux distros. Prefer wheels for speed/reproducibility; publish both when extensions are present.

232. How do you create a CLI entry point for a package?

Declare console_scripts entry points in pyproject.toml so installers create executables. For single-file tools, zipapp or shiv/pex can bundle code and deps into a self-contained archive.

233. What are logging best practices in Python?

Use the stdlib logging with a module-level logger, structured context (extra or structlog), consistent levels, UTC timestamps, JSON handlers in prod, and a dictConfig. Never use print for operational logs.

234. How do you add observability (metrics/tracing) to a Python service?

Expose metrics (Prometheus client), propagate trace IDs via contextvars, and instrument HTTP/DB with OpenTelemetry. Correlate logs, metrics, traces using the same request ID.

235. What security pitfalls should you avoid?

Never eval/pickle untrusted data; use safe YAML loaders; store secrets in env/secret managers; validate inputs (Pydantic), pin deps, and run SCA tools. Keep least privilege for file/network access.

236. How do you avoid N+1 queries in SQLAlchemy?

Use eager loading (selectinload/joinedload), write explicit select() joins, and profile with echo/query counters. Manage sessions per request and batch operations where possible.

237. What are Django ORM performance tips?

Use select_related for FK joins, prefetch_related for M2M, annotate/aggregate on the DB, and avoid per-row .save() loops (bulk ops). Wrap logic in transactions and watch migration order.

238. How do you work with async databases in Python?

Use async drivers (e.g., asyncpg) or SQLAlchemy asyncio with async engines/sessions, connection pools, and async with transactions. Keep CPU work off the event loop.

239. What caching strategies are common in Python apps?

In-process memoization (functools.lru_cache), request-scoped caches, and shared caches (Redis) with TTL/tags. Design invalidation carefully; prefer cache-aside or write-through depending on consistency needs.

240. How do you run background jobs from a web app?

Use task queues like Celery/RQ/Huey. Ensure idempotency, retries with backoff, and visibility timeouts. Pass small payloads (IDs), fetch data in the worker, and monitor queue lag.

241. What advanced pytest features are most useful?

Parameterized tests, fixture scopes (function/module/session), monkeypatch, caplog, custom markers, and xdist for parallelism. Keep tests deterministic and fast.

242. How do you test HTTP integrations effectively?

Use responses/pytest-httpserver or httpx's test client to stub endpoints. Prefer contract tests for third-party APIs and record/replay with VCR-style tools when appropriate.

243. Explain NumPy broadcasting and why it's fast.

Broadcasting applies operations across arrays with compatible shapes without explicit Python loops. Speed comes from vectorized C loops and contiguous memory access using strides.

244. How do you speed up pandas workloads?

Prefer vectorized ops over apply, choose efficient dtypes (categorical, nullable), avoid chained indexing, process in chunks for large data, and consider Polars for query-engine style speed.

245. What are key PySpark concepts for Python users?

Use DataFrames (Catalyst optimizes) over RDDs, remember transformations are lazy, and understand narrow vs wide dependencies (shuffle). Minimize UDFs; when needed use Pandas/Arrow UDFs.

246. How do Dask and Ray help with Python scalability?

They distribute Python workloads across cores/machines. Dask focuses on dataframes/arrays and task graphs; Ray provides a general actor/task model and ecosystem (Ray Tune/Serve).

247. When do Numba or Cython provide speedups?

They accelerate numeric loops by compiling to machine code. Best for tight loops on NumPy arrays with simple types. Numba's nopython mode avoids Python objects; Cython allows gradual static typing.

248. What is the buffer protocol and memoryview?

The buffer protocol exposes raw memory of objects (bytes/bytearray/array/NumPy) without copying. memoryview lets you slice and manipulate large binary data zero-copy, improving I/O performance.

249. ## How do you write robust async HTTP clients?

Use httpx/aiohttp with connection pooling, timeouts, and retries with exponential backoff. Respect rate limits, handle 429/5xx, and propagate cancellation to avoid orphan tasks.

250. ## What are practical deployment tips for Python services?

Package as wheels, build slim Docker images, pin deps, run health checks, and choose the right server (Gunicorn+Uvicorn workers for ASGI). Use pre-fork for CPU-bound and async workers for IO-bound workloads.

251. ## How do Python import hooks work?

Imports are handled by finders on sys.meta_path that return a module spec, and loaders that create/execute modules from that spec (via importlib). You can insert custom finders/loaders to load modules from databases, encrypted files, or remote sources.

252. ## What are namespace packages and when would you use them?

Namespace packages (PEP 420) allow a single logical package to be spread across multiple directories/distributions without an init.py. They're useful for plugin ecosystems or splitting a large package into separately installed parts.

253. ## What is an editable install and how does it differ from a regular install?

Editable installs (PEP 660) let you import your project directly from the source tree while still using the build backend, so changes reflect without reinstalling. A regular install copies built artifacts into site-packages.

254. ## Explain absolute vs relative imports and recommended practice.

Absolute imports reference from the project root (package.module), while relative imports use leading dots to traverse the package hierarchy. Prefer absolute imports for clarity and fewer surprises; use relative imports for tightly coupled subpackages.

255. ## What does all do in a module?

all controls which names are imported when using from module import * and serves as a public API hint for tools. It doesn't restrict direct attribute access but clarifies intended exports.

256. ## Which advanced dataclass features are most useful?

frozen=True for immutability, slots=True to reduce memory, kw_only=True to force keyword args, field(default_factory=...) for dynamic defaults, and __post_init__ for validation/derived fields.

257. ## Dataclasses vs attrs vs Pydantic—how do they compare?

dataclasses: lightweight stdlib modeling. attrs: older, very feature-rich with converters/validators. Pydantic: validation and parsing of external data with types (v2 uses pydantic-core), great for API schemas/config.

258. ## What are Protocols and structural subtyping?

Protocols (typing.Protocol) define method/attribute shapes that a type can satisfy implicitly (duck typing with types). Unlike ABCs, they don't require inheritance; any object with the right members matches.

259. ## Explain variance (covariant/contravariant/invariant) in Python typing.

TypeVars can be covariant (T_co), contravariant (T_contra), or invariant (default). Covariant allows substituting subtypes (e.g., sequences), contravariant for consumer-only types (e.g., callback params), invariant when both read/write occur.

260. ## When should you use TypedDict, Final, ClassVar, and Literal?

TypedDict types JSON/dict-like structures, Final marks names not to be reassigned/overridden, ClassVar denotes class-only attributes ignored by instance typing, and Literal constrains values to specific constants.

261. ## What are exception groups and except* used for?

ExceptionGroup lets you raise/handle multiple exceptions together (e.g., from concurrent tasks). except* handles members of the group by type, enabling granular error processing in async/concurrent code.

262. ## How do ExitStack and AsyncExitStack help with resource management?

They compose multiple context managers dynamically, ensuring proper teardown even when the set of resources isn't known upfront. AsyncExitStack does the same for async context managers.

263. ## When should you use weakref and why avoid __del__?

weakref lets you reference objects without increasing refcounts (useful for caches/graphs). __del__ complicates GC for cycles and may not run at exit; prefer weakref.finalize for safe cleanup callbacks.

264. ## Explain the descriptor protocol and how property uses it.

Descriptors implement __get__, __set__, __delete__, and optionally __set_name__ to control attribute access. property is a descriptor providing managed attributes (getters/setters/deleters) with natural syntax.

265. ## What are the pros and cons of __slots__?

__slots__ removes per-instance dict to save memory and speed attribute access. Trade-offs: no dynamic attributes, some MRO/multiple inheritance caveats, and pickling needs care unless using defaults.

266. ## How do Python memory leaks typically occur and how do you diagnose them?

Leaks often come from unexpected object retention: reference cycles with __del__, global caches, lru_cache growth, or lingering tasks. Use tracemalloc, objgraph, and heap snapshots; fix by breaking cycles and bounding caches.

267. ## Compare multiprocessing start methods: fork, spawn, forkserver.

fork is fast but can copy unsafe state on Unix; spawn starts a fresh interpreter (portable, safer for threads); forkserver pre-forks a clean server to launch children. Choose based on safety vs startup cost.

268. ## How do you share data efficiently between processes?

Use multiprocessing.shared_memory for zero-copy NumPy buffers, Array/Value for small C types, or a Manager for proxied Python objects (slower). Avoid pickling large payloads repeatedly—share by reference when possible.

269. ## What are best practices for running subprocesses?

Prefer subprocess.run with explicit timeouts and check=True, capture streams carefully to avoid deadlocks, and sanitize inputs. For async, use asyncio.create_subprocess_exec and await pipes.

270. ## How can you optimize file I/O in Python?

Use buffered I/O, iterate in chunks, memory-map large files with mmap for random access, and avoid per-line Python overhead where possible. Prefer pathlib.Path APIs for clarity and portability.

271. ## What tools and strategies help with profiling Python code?

Start with cProfile + pstats/SnakeViz for hotspots, use time.perf_counter for micro-benchmarks, and sampling profilers for low overhead. Profile realistic inputs and isolate I/O from CPU hotspots.

272. ## How do you package and load non-code data files?

Declare them in the build config and access at runtime via importlib.resources (files()/as_file()) instead of relying on file paths. This works for wheels, zip imports, and packaged resources.

273. ## How can you speed up (de)serialization of JSON?

Minimize conversions, use json.loads/dumps with separators for compact output, and pre-validate shapes. For heavy workloads, consider faster third-party libraries and avoid per-object hooks when possible.

274. ## argparse vs Click vs Typer—when to choose which?

argparse is stdlib and dependable for basic/complex CLIs; Click offers a nicer developer experience with decorators and colorized help; Typer builds on Click with type hints and automatic help generation.

275. ## What should you know about text encodings and i18n in Python?

Always open files with encoding="utf-8" and handle errors explicitly; be aware of the OS default encoding. Normalize Unicode when comparing, and use gettext/formatting libraries for localization.

276. ## What is pyproject.toml and why is it important?

pyproject.toml is a build and project configuration file (PEP 518/517/621). It declares the build backend (e.g., setuptools, hatchling, poetry), project metadata (name, version, dependencies), and tool configs. It standardizes builds and reduces setup.py/setup.cfg fragmentation.

277. ## What is a Python wheel (.whl) file?

A wheel is a built distribution format that contains ready-to-install artifacts. Installing wheels avoids building from source, speeding installs and making them more reproducible. Platform/ABI tags on the filename indicate compatibility.

278. ## How do you create and use a virtual environment?

Run python -m venv .venv to create one, then activate it (source .venv/bin/activate on Unix, .venv\Scripts\activate on Windows). Install packages with pip inside the venv to isolate dependencies per project.

279. ## How should you manage configuration and environment variables?

Read from the environment using os.environ and provide defaults. Keep secrets out of code, pass via env or secret managers. Optionally use a loader (e.g., python-dotenv) and validate with dataclasses or Pydantic models.

280. ## What is the purpose of the contextvars module?

contextvars provides context-local state that works with async tasks and concurrency. It lets you store request-scoped data (e.g., correlation IDs) without relying on thread-local storage, which breaks in async code.

281. How do you cancel tasks safely in asyncio?

Call task.cancel() and then await the task to let it handle CancelledError. Use asyncio.wait_for to add timeouts. Wrap critical sections with asyncio.shield() if you don't want them cancelled.

282. What is backpressure in async I/O and how do you handle it?

Backpressure occurs when producers outpace consumers. Use bounded queues, await writer drain() on streams, throttle producers, and design protocols that allow consumers to signal capacity.

283. When should you choose threading, multiprocessing, or asyncio?

Use threads for I/O-bound concurrency with blocking libraries; multiprocessing for CPU-bound workloads (bypasses the GIL via processes); asyncio for high-concurrency I/O when libraries are async-aware.

284. How do context managers work and how do you build one?

Context managers implement __enter__/__exit__ to acquire/release resources. Create with a class or with contextlib.contextmanager decorator yielding a resource and handling cleanup in a try/finally.

285. Differentiate @staticmethod, @classmethod, and @property.

staticmethod is a namespaced function with no implicit first arg; classmethod receives the class as cls and is useful for alternative constructors; property is a descriptor for computed attributes with getter/setter.

286. What is method resolution order (MRO) and how does super() use it?

MRO is the linearization of class inheritance determining attribute lookup order (C3 linearization). super() follows the MRO to call the next method, enabling cooperative multiple inheritance.

287. How do you implement iterables and iterators?

An iterable defines __iter__ returning an iterator. An iterator defines __next__ and raises StopIteration when exhausted. Generators automatically implement the iterator protocol.

288. Which itertools functions are most useful?

chain, islice, product, permutations, combinations, groupby, accumulate, and tee help build memory-efficient pipelines for combinatorics, slicing, grouping, and running totals.

289. When should you use functools.lru_cache?

Use it to memoize pure functions to avoid recomputation. It's effective for expensive deterministic calls with a small input domain. Size the cache and consider invalidation for changing data.

290. Explain naive vs aware datetimes and how to handle time zones.

Naive datetimes lack tzinfo; aware datetimes include a timezone. Use datetime.now(ZoneInfo('UTC')) or zoneinfo (stdlib) to attach time zones. Store in UTC, convert for display, and beware DST transitions.

291. When should you use decimal instead of float?

Use decimal for exact base-10 arithmetic (currency, financial apps) to avoid binary floating-point rounding issues. It's slower but configurable with contexts for precision and rounding modes.

292. What features does structural pattern matching (match) provide?

It matches literals, sequences, mappings, classes (by attributes), and supports guards (if). It enables declarative branching but isn't exhaustive by default—ensure a catch-all case for safety.

293. How do you use typing.overload correctly?

Write multiple @overload stubs with precise signatures and no bodies, then a single implementation handling all cases. Type checkers use the stubs to infer return types based on input signatures.

294. What is functools.singledispatch and when is it useful?

singledispatch enables function overloading based on the first argument's type at runtime. It's helpful for implementing type-specific behavior without large if/elif trees or altering classes.

295. How do you test asynchronous code in Python?

Use an async test runner or plugin (e.g., pytest with pytest-asyncio). Mark tests with @pytest.mark.asyncio and await coroutines. For timeouts, cancellations, and tasks, assert behavior and exceptions explicitly.

296. How do you mock dependencies in tests?

Use unittest.mock to patch objects where they’re looked up, not defined. Mock/MagicMock provide fakes; patch replaces names within a context; spec enforces attribute correctness; side_effect simulates errors.

297. How can you apply dependency injection in Python?

Pass collaborators via constructor or parameters, prefer protocols/ABCs for boundaries, and use factories/config objects. Keep side effects at edges and inject I/O, making core logic pure and testable.

298. Is pickling safe? How should you handle untrusted data?

Pickle can execute arbitrary code on load, so never unpickle untrusted data. Use safe formats like JSON/msgpack, or a restricted, audited deserializer. For trusted use, pin protocol versions and classes.

299. What are best practices for logging in Python?

Use the logging module with module-level loggers, structured keys in messages, and central configuration (dictConfig). Set appropriate levels, avoid printing directly, and propagate to handlers (file/STDOUT).

300. What’s the difference between __getattr__ and __getattribute__?

__getattribute__ intercepts all attribute access; use sparingly and call object.__getattribute__ to avoid recursion. __getattr__ is a fallback only when normal lookup fails, useful for dynamic or deprecated attributes.

Tip: Sketch broker/partition diagrams when answering replication or rebalancing questions.

SEARCH

Search topics... Go

INTERVIEW TOPICS

Java	Q&A
Java 8	Q&A
Spring	Q&A
Hibernate	Q&A
React JS	Q&A
SQL	Q&A
MongoDB	Q&A
JavaScript	Q&A
Kafka	Q&A
Design Patterns	Q&A
Microservices	Q&A
REST API	Q&A
Behavioral	Q&A

QUICK ACTIONS

Start System Design | Practice Coding | Earn a Certificate

RECOMMENDED READING

- Learn Java 17 Programming
- Learn JavaScript Quickly
- SQL QuickStart Guide
- System Design Interview

See all ➜

SPONSORED

Ads help keep content free.

NEED HELP?

Have a topic request or correction? We'd love to hear from you.

Contact Us

THANK YOU!

Your comments and complements are always the inspiration that keeps us motivated.

ABOUT

PROJECTS

ABOUT US

BLOG

AWARDS

ADDRESS

Sterling, Virginia, 20166, US

info@techventurellc.com

641 451 3463

641 451 3463

FOLLOW US