🔰 Tier 1 — Beginner (0–1 year) — Questions 1–50

Focus: syntax, basic data structures, functions, control flow, simple OOP, file I/O.

1. Explain Python's execution model for a simple script (how the interpreter runs `.py` file).
2. List Python's core built-in data types and give a one-line example for each.
3. Explain the difference between `==` and `is` with a short example.
4. Why are Python strings immutable? Give two practical consequences of immutability.
5. Explain list, tuple, set, and dict in terms of mutability and typical use-case.
6. Show how to iterate over a list and also with index (two methods). **[CODE]** — write a function `enumerate_items(lst)` that returns a list of `"(index) value"` strings.
7. Explain how slicing works (start:stop:step). Give an example reversing a list with slicing.
8. What are list comprehensions? Convert a `for` loop that squares numbers into a list comprehension. **[CODE]** — implement `squares(n)` returning `[0..n-1]` squared.
9. Differentiate `append`, `extend`, and `insert` with examples.
10. How do you merge two dictionaries in Python 3.9+? Show at least two ways. **[CODE]** — `merge_dicts(a,b)` returns merged dict without modifying originals.
11. Explain exception handling: `try/except/finally/else`. Provide an example that reads an integer from input with exception handling. **[CODE]** — `safe_int(s)` returns `int` or `None`.
12. What is a function, and what does `return` do? What happens if a function has no `return`?
13. Explain default parameter values and the mutable-default gotcha (show example bug). **[CODE]** — fix the bug in `def add_item(x, item=[]):` to be safe.

14. What is variable scope? Define LEGB and give one example showing local vs global.

15. What is `__name__ == "__main__"` used for? Give an example script that can be imported or run.

16. Explain `*args` and `**kwargs` with an example function that accepts both. **[CODE]** — implement `build_url(base, **params)` that returns URL with query string.

17. What is a module and a package? How do you import from them?

18. Explain `help()` and `dir()` built-ins and when you'd use them.

19. What does the `with` statement do for file handling? Why prefer it? **[CODE]** — write `read_lines(path)` returning list of lines safely.

20. How do you open a file in binary mode and why would you? Give an example reading bytes.

21. Explain `str()`, `repr()`, and when each is used.

22. What's the difference between `pop()` and `remove()` on lists? Show examples.

23. How do you check the type of an object? Compare `isinstance()` vs `type()`.

24. What are Python boolean values (`True`/`False`)? How are other types truthy/falsy? Provide examples.

25. What is a dictionary comprehension? Convert a list of pairs to dict with a comprehension. **[CODE]** — `pairs_to_dict(pairs)`.

26. Explain how to format strings with f-strings, `.format()`, and `%`. Show an example using f-strings.

27. What are multi-line strings and triple quotes used for? Show use for docstring.

28. How do you create and use a simple class with one attribute and one method? **[CODE]** — implement `class Person` with `name` and `greet()`.

29. What are instance variables vs class variables? Show an example where class variable is shared across instances.

30. Explain `__init__` and what it does in a class.

31. How does inheritance work? Write a base class `Animal` and subclass `Dog` overriding a method. **[CODE]** — implement both and show polymorphism with a shared method name.

32. What is a docstring and how does it differ from comments? How do you access it at runtime?

33. Explain the difference between `for` and `while` loops with examples.

34. How do you break and continue inside loops? Show examples.

35. What are boolean operators `and`, `or`, `not` — show precedence example.

36. Explain the `range()` function and typical use in loops. How is `range(0)` different from `[]`?

37. How do you convert between data types like `str` → `int` → `float` safely (with exception handling)? **[CODE]** — `to_int(s)` returning tuple `(ok, int_val_or_None)`.

38. Explain `set` and basic set operations: union, intersection, difference. **[CODE]** — `unique_items(lst)` using `set`.

39. How to use `enumerate()` and `zip()` — show examples and when they're useful.

40. What is the Python REPL and how can it aid learning/debugging? Mention `python -i` and `ipython` briefly.

41. How do you install a package with pip and import it? Explain virtualenv at a basic level.

42. Describe built-in `map`, `filter`, and `reduce` — when to use vs list comprehensions. **[CODE]** — `even_squares(n)` using `map`/`filter`.

43. What are lambda functions and where are they useful? Show short example sorting by a key using a lambda.

44. Explain how to test a script via `pytest` or simple asserts — show a small assert example for a function. **[CODE]** — write simple test assert for `squares(5)`.

45. What is JSON and how do you parse/serialize JSON in Python? **[CODE]** — `load_json(path)` returns parsed object.

46. Explain what an interpreter error vs exception is; give examples of SyntaxError and ValueError.

47. How do you use comments and when to prefer inline comments vs docstrings?

48. What is the `os` module used for? Provide an example listing files in a directory. **[CODE]** — `list_py_files(path)`.

49. Explain simple debugging techniques: print debugging, using `pdb.set_trace()` or `breakpoint()`.

50. Describe how to read environment variables in Python and one use-case (e.g., configuration secrets). **[CODE]** — `get_env_or_default(key, default)`.

🟡 Tier 2 — Junior (1–2 years) — Questions 51–100

Focus: intermediate OOP, modules, packaging, testing, stdlib, basic algorithms, data handling.

51. Explain the difference between shallow and deep copy; show `copy.copy` vs `copy.deepcopy`. **[CODE]** — demonstrate the difference with nested lists.

52. What is duck typing? Give an example where duck typing is useful.

53. Explain `@staticmethod` vs `@classmethod` vs instance methods, with use-cases. **[CODE]** — class `Counter` with classmethod constructor.

54. How does Python implement private attributes (name mangling)? Show an example using `__private`.

55. What is an abstract base class (ABC) and how do you define/force implementation of methods? **[CODE]** — define an ABC with `abc` module.

56. Explain multiple inheritance and the Method Resolution Order (MRO). Give an MRO example diamond problem and how Python resolves it.

57. How do you make an object iterable? Implement `__iter__` and `__next__`. **[CODE]** — implement simple `RangeLike` iterator class.

58. What are generators (`yield`) and generator expressions? When prefer generators? **[CODE]** — write a generator `fibs()` yielding Fibonacci numbers.

59. Compare generator functions and coroutines at a high level.

60. Explain decorators: basic decorator that logs function entry/exit. **[CODE]** — implement `@log_calls` decorator preserving `__name__` & `__doc__`.

61. How do you write parameterized decorators (decorator factory)? Provide an example.

62. Explain function annotations (type hints) and how they're used with tools like `mypy`. Show simple annotated function.

63. What is `typing.Optional`, `Union`, `List`, `Dict` — show small examples.

64. How do you write and run unit tests with `pytest`? Show a minimal test file layout. **[CODE]** — write a pytest function for `fibs()` returning first 5 numbers.

65. Explain `unittest.mock` basics: patching functions and mocking objects. **[CODE]** — mock a function that calls external API.

66. How do you log in Python? Explain `logging` module basics and logging levels. **[CODE]** — configure basic logger writing to file.

67. Explain context managers and `contextlib.contextmanager` decorator. **[CODE]** — write a custom context manager that times a code block.

68. How do you read/write CSV files? Show `csv` module usage and `pandas` brief mention. **[CODE]** — `count_rows_csv(path)`.

69. What are list/tuple/dict performance trade-offs for membership tests, indexing, and iteration? Provide Big-O where relevant.

70. Explain how `heapq` works and one use-case (priority queue). **[CODE]** — use `heapq` to merge k sorted lists.

71. What are namedtuples and `dataclasses` — when to use which? **[CODE]** — create a `@dataclass Point(x,y)` and a method distance.

72. Explain basic algorithmic complexity: O(1), O(n), O(n log n), O(n^2). Give one example algorithm for each complexity.

73. How to profile a function's runtime using `timeit` and `cProfile`. **[CODE]** — demonstrate `timeit` to compare two implementations.

74. Explain the `pathlib` module and how it improves on `os.path`. **[CODE]** — implement `find_files_with_ext(dir, ext)`.

75. What is `itertools` and name 4 useful functions with short examples (`chain`, `islice`, `product`, `groupby`).

76. How does Python's `sorted()` accept a key function? Show sorting a list of dicts by key. **[CODE]** — `sort_by_key(list_of_dicts, keyname)`.

77. Explain JSON vs pickle — use-cases and security implications of pickle.

78. Describe how to work with timestamps and dates using `datetime`. Parse and format a timestamp string. **[CODE]** — `parse_iso(ts)` → `datetime`.

79. Explain the Global Interpreter Lock (GIL) conceptually and why CPU-bound tasks aren't helped by threads in CPython.

80. How does `multiprocessing` differ from `threading`? Show a small `multiprocessing.Pool` map example. **[CODE]** — parallel `square` computation.

81. How do you implement caching/memoization? Show `functools.lru_cache` usage. **[CODE]** — memoize Fibonacci.

82. Explain the role of virtual environments and how to create one with `venv`.

83. How to publish a package to PyPI at a high level (setup files, versioning, packaging).

84. Explain semantic versioning (MAJOR.MINOR.PATCH) and why it matters for packages.

85. How to safely handle secrets and config in code (env vars, config files). **[CODE]** — load config from `~/.config/app.json`.

86. Explain how to use `subprocess` safely for running shell commands. Show avoiding shell injection risks. **[CODE]** — run `ls` safely capturing output.

87. How do you use type checking with `typing.Protocol` to express structural typing? Provide a small protocol example.

88. What is continuous integration (CI) and how would you test Python code on CI (e.g., GitHub Actions basics)?

89. How to use `sqlite3` built-in module for simple local DB operations: create table, insert, select. **[CODE]** — simple `create_and_insert()` demonstrating commit.

90. Explain the differences between `==` and `__eq__` implementation for objects; implement `__eq__` for a class. **[CODE]** — class `Point` with `__eq__`.

91. How do you safely handle file encodings (UTF-8) when reading/writing? Discuss common pitfalls.

92. What is a memory leak in Python and common causes (e.g., lingering references, caching) and how to discover them?

93. Explain how to use `pdb` for stepping through code and watching variables. Provide sequence of commands to set a breakpoint and continue.

94. How to implement a CLI utility with `argparse` (positional and optional arguments). **[CODE]** — small tool `grep_like.py` that takes `--pattern` and `path`.

95. What are Python entry points and console_scripts for creating CLI tools with packaging?

96. Explain how to handle binary data and struct packing/unpacking with `struct`. **[CODE]** — pack two ints and unpack.

97. Explain how to perform HTTP requests with `requests` module and basic error handling/timeouts. **[CODE]** — fetch JSON with timeout and handle errors.

98. How to write clean code: naming, single responsibility, small functions — give three rules you follow in practice.

99. Explain the role of code reviews and linters like `flake8` and formatters like `black`. How to add them to a project.

100. Provide a brief plan to migrate Python 2 code to Python 3 (key areas to check).

🔵 Tier 3 — Mid-Level (2–5 years) — Questions 101–150

Focus: architecture, async, performance, testing at scale, design patterns, databases, networking.

101. Explain asynchronous programming: event loop, tasks, coroutines, futures in `asyncio`.

102. How do `async def` and `await` work? Show converting a blocking function to `async`. **[CODE]** — wrap `time.sleep` to non-blocking using `asyncio.sleep`.

103. When use `asyncio` vs `threading` vs `multiprocessing`? Provide example use-cases.

104. Explain TCP vs UDP fundamentals and how to create a simple TCP client/server with `socket`. **[CODE]** — small echo server and client.

105. What are design patterns useful in Python: Singleton, Factory, Strategy — implement Strategy pattern with functions. **[CODE]** — `Strategy` selecting different sort strategies.

106. How to implement dependency injection in Python for testability? Provide an example with constructor injection. **[CODE]** — service class injected with database client.

107. Explain how to structure a medium-sized Python project: packages, tests, docs, scripts, config. Provide a sample directory layout.

108. Describe transactional behavior in databases and how to manage transactions in Python (e.g., commit/rollback). **[CODE]** — `sqlite3` transaction with rollback on exception.

109. How do ORMs work? Give example mapping with SQLAlchemy basic model and simple query. **[CODE]** — define `User` model and query by `id`.

110. Explain connection pooling and why it matters for DB-heavy apps. Mention `psycopg2` or DB driver concepts.

111. How to write maintainable async code and avoid anti-patterns like blocking the event loop. Give examples of common pitfalls.

112. What are websockets and how to implement a basic websocket handler (conceptually) with `websockets` or `aiohttp`. **[CODE]** — simple websocket echo server pseudocode.

113. Explain REST principles and how to build a small REST API with `FastAPI` or `Flask`. **[CODE]** — minimal `FastAPI` endpoint returning JSON.

114. How to implement rate limiting for an API and one strategy to do so (token bucket). Provide pseudo-implementation. **[CODE]** — simple in-memory rate limiter.

115. Explain authentication flows: basic auth, token auth (JWT), OAuth2 high-level. Show how to validate a JWT in Python. **[CODE]** — decode/verify JWT using `pyjwt` (pseudocode).

116. How to design for observability: metrics, logging, tracing. Provide examples of metrics to collect.

117. Explain message brokers (RabbitMQ, Kafka) conceptually and when to use them/advantages. How to publish/consume with a client library at high-level. **[CODE]** — pseudocode publishing to Kafka.

118. How to implement background job workers with `celery` or `rq` — job queue concept and result handling.

119. What are race conditions and deadlocks? Provide an example in threading and how to prevent them. **[CODE]** — demonstrate race on shared counter and fix with `Lock`.

120. How to perform advanced profiling (memory and CPU) for a web app under load — mention `pyinstrument`, `memray`, `tracemalloc`.

121. Explain how to optimize algorithmic performance: identify hotspots, use C-accelerated libs (NumPy), caching, and lazy evaluation.

122. How to design idempotent APIs and why idempotency matters in distributed systems.

123. Explain eventual consistency vs strong consistency in distributed stores and one approach to achieve consistency.

124. How to implement schema migrations for databases (Alembic, Django migrations) and why they matter. **[CODE]** — simple Alembic command example (pseudocode).

125. How to manage secrets at scale: vaults (HashiCorp Vault), KMS, env variables on deployment — tradeoffs.

126. What is backpressure in distributed messaging and how to handle it in a consumer.

127. How to implement data validation and serialization (e.g., Pydantic) and benefits for API development. **[CODE]** — Pydantic model example for `User`.

128. Explain the CAP theorem and how it applies to system design choices.

129. How to manage configuration across environments (12-factor config) and one implementation approach using env + config lib. **[CODE]** — sample config loader merging env and file.

130. How to write contract tests for microservices and why they are useful. Provide example of consumer-driven contract test at high-level.

131. Explain caching strategies: write-through, write-back, cache-aside. Provide scenario where cache-aside suits best.

132. How to design a pagination API that is safe and performant for large datasets (cursor vs offset-based). **[CODE]** — implement offset pagination function stub.

133. How to handle large file uploads efficiently in a web service (streaming, chunking). Describe memory issues to avoid.

134. Explain how to use async database drivers vs sync drivers and tradeoffs with ORMs.

135. What is horizontal scaling vs vertical scaling? Give Python app examples of each strategy.

136. How to design health checks and readiness/liveness endpoints for services. **[CODE]** — simple `/health` endpoint returning status.

137. Explain how to run and test asynchronous code in pytest (using `pytest-asyncio`). **[CODE]** — async test sample.

138. How to do blue-green deployments or canary releases for a Python web app (high-level).

139. Explain how to implement feature flags and rollout strategies; mention `flipper` or `unleash` style client usage.

140. How to maintain backward compatibility in APIs and strategies to version APIs safely.

141. Explain content negotiation and how to return different representations (JSON/CSV) from an endpoint. **[CODE]** — implement a view that returns CSV if `Accept` header set.

142. What is message deduplication and how to ensure idempotent message processing? Example approach.

143. How to design export jobs for large data (ETL) and considerations like batching, backpressure, retries.

144. Explain how to build a basic CLI tool with `click` that has subcommands and options. **[CODE]** — `cli.py` with `init` and `run` commands.

145. How to use property decorators `@property` to expose computed attributes; show setter/deleter example. **[CODE]** — `@property` full example.

146. Explain how to structure and write integration tests that involve DB and external services — use fixtures and test containers.

147. How to use `concurrent.futures` for thread/process pools and when it simplifies code compared to manual threading. **[CODE]** — use `ThreadPoolExecutor` to parallelize I/O calls.

148. How to design retry/backoff strategies for transient failures; implement exponential backoff pseudocode. **[CODE]** — retry decorator with backoff.

149. Explain TLS/HTTPS basics and how to configure a Python web server to serve over TLS (conceptually).

150. How to do blueprints/modules in Flask or routers in FastAPI to keep code modular and testable. Provide short example module layout.

🔴 Tier 4 — Senior (5+ years) — Questions 151–200

Focus: deep internals, optimizations, language internals, large-system design, mentorship/leadership technical decisions, security, deployment.

151. Explain CPython's memory model: object headers, reference counting, arenas and pools (conceptually).

152. How does Python's garbage collector interact with reference counting to collect cycles? Provide an example where `__del__` complicates collection.

153. Explain how Python dictionaries are implemented (hash table details) and why insertion order is preserved since 3.7.

154. What is interning of strings and small integer caching? How does it improve performance?

155. Explain Python bytecode lifecycle: source → AST → bytecode → interpreter loop. How to disassemble with `dis`. **[CODE]** — disassemble simple function.

156. How do `__slots__` work internally and when to use them for memory optimization? Provide example and caveats. **[CODE]** — class with `__slots__`.

157. Explain the Global Interpreter Lock (GIL) internals and how it affects multi-threaded programs in CPython. Mention efforts like subinterpreters and GIL removal proposals.

158. What are extension modules in C (CPython C-API) and when to write one vs using Cython? Give tradeoffs.

159. How to profile and optimize memory usage of a long-running process — tools and approaches (tracemalloc, heapy, objgraph). **[CODE]** — sample `tracemalloc` snapshot usage.

160. Explain how to implement a C extension that speeds up a tight loop (high-level steps).

161. How to write Python that takes advantage of vectorized NumPy operations instead of Python loops; show performance reasoning. **[CODE]** — implement vectorized sum vs Python loop.

162. Explain JIT options (PyPy) and tradeoffs between PyPy and CPython. When consider switching runtime.

163. How to perform zero-downtime migrations for database schema changes in production — concrete steps and risky operations to avoid.

164. Explain advanced concurrency patterns: actor model, work stealing, and how they could be implemented in Python at high level.

165. How to harden a Python application for security: input validation, SQL injection prevention, XSS, CSRF mitigations (for web apps). Provide concrete code practices.

166. How to do threat modeling for a Python-based web service and list top 5 attack vectors.

167. Explain supply-chain attacks (typosquatting on PyPI) and mitigation strategies (pinned dependencies, lockfiles, verifying package signatures).

168. How to build a robust deployment pipeline: staging/testing/production, automatic rollback, migrations, health checks. Provide CI/CD steps.

169. How to design an observability stack: metrics aggregation, alerting thresholds, tracing with OpenTelemetry. Give example metrics for a Python API.

170. Explain advanced caching: cache invalidation strategies in distributed caches and tradeoffs (consistency vs performance).

171. How to implement consistent hashing for distributed caching or sharding (conceptual algorithm & use-case). **[CODE]** — simple consistent-hash ring stub.

172. How to implement custom serialization formats for performance (msgpack, protobuf) and tradeoffs vs JSON. **[CODE]** — serialize/deserialize with `msgpack`.

173. Explain how to debug production incidents: collecting core dumps, log aggregation, post-mortem practices. Provide checklist steps.

174. How to measure and reduce tail latency in an application; give techniques like hedged requests and timeouts.

175. Discuss backfill and recomputation strategies for data pipelines — idempotence and incremental run strategies.

176. How to use advanced `multiprocessing` patterns safely (shared memory, `multiprocessing.Manager`, `ForkServer`) and pitfalls on Mac/Windows. **[CODE]** —

pool with initializer.

177. Explain how to design a system for multi-tenant isolation in a SaaS application; mention DB, caching and filesystem isolation strategies.

178. How to approach technical design interviews as a hiring manager: what to evaluate, pitfalls, rubric example.

179. How to mentor mid-level engineers — give a 30/60/90 plan for their growth focusing on coding, design, and ownership.

180. Explain running Python in constrained environments (embedded, serverless) and tradeoffs for cold starts, package size.

181. How to reduce startup time of a Python app (faster imports, lazy imports, compiled wheels). **[CODE]** — show lazy import pattern.

182. Explain database indexing internals and when an index hurts write performance; give SQL tuning tips.

183. How to design and test rolling upgrades across heterogeneous clusters maintaining compatibility.

184. Explain advanced type-system usage: generics, `TypedDict`, `ParamSpec`, and how they help in large codebases. **[CODE]** — `TypedDict` example for request payload.

185. How to handle long-running job orchestration (Airflow, Prefect): retries, task dependencies, idempotency. Provide DAG design considerations.

186. How to design a secure plugin system allowing third-party code without compromising host: sandboxing options and limitations.

187. How to benchmark and compare performance across Python versions or builds; what to control for and reproducibility tips. **[CODE]** — microbenchmark harness using `timeit` and `perf`.

188. Explain how to build a Python-based library that must maintain binary compatibility across minor versions (C ABI concerns).

189. How to evaluate and choose third-party libraries (criteria: maintenance, security, API stability, performance). Provide checklist.

190. How to implement graceful shutdowns in multi-threaded or async Python services responding to SIGTERM/SIGINT. **[CODE]** — skeleton graceful shutdown handling.

191. Explain advanced packaging: building wheels, manylinux, binary dependencies, and the role of `pyproject.toml`.

192. How to handle GDPR/data deletion requirements in a system: identify PII, implement deletion workflows, audit trails.

193. How to do load-testing and chaos testing safely (locust, k6) and how to interpret results to inform capacity planning. **[CODE]** — simple locust user behavior example (pseudocode).

194. Explain the economics of cloud usage for Python apps (serverless vs containers vs VMs) and cost-saving strategies (right-sizing, spot instances).

195. How to design for observability at scale: metrics cardinality limits, sample rates, high-cardinality dimensions. Provide mitigation strategies.

196. How to run Python with native threads for CPU-bound work using subprocessing patterns (workers) and efficient IPC. **[CODE]** — worker pool with `multiprocessing` and queue.

197. Explain how to design and manage a monorepo vs multiple repos — build/test implications for Python projects.

198. How to assess technical debt and plan remediation vs delivering features — provide triage framework and acceptance criteria.

199. How to set up a secure build pipeline ensuring deterministic builds and artifact signing. Provide high-level steps.

200. Give a personal technical leadership plan: how would you improve architecture, team practices, quality and reliability over the next 12 months for a Python service you own? Provide 6 concrete initiatives with measurable outcomes.

✅ How I'll use your answers (and next steps)

- You said you'll answer — awesome. Send me answers (one-by-one, batches, or the coding solutions).
- For **coding tasks** I'll run quick checks (pseudo-grading) and give:
    - Correctness feedback
    - Time/space complexity analysis
    - Improvements and alternative approaches
    - Micro-optimizations or design feedback