Python Best Practices

Python is a high-level, interpreted programming language known for its readability and versatility. Following best practices ensures maintainable, efficient, and robust code.

Type hints improve code readability and enable static analysis. Use typing module for complex types like Optional, Union, List, Dict, and custom generics. Tools like mypy and pyright enforce type correctness.

Virtual environments isolate project dependencies. Tools include venv (built-in), virtualenv, conda, and poetry. Always pin dependency versions in requirements.txt or pyproject.toml for reproducible builds.

The SOLID principles apply to Python: Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion. While Python is flexible, these principles guide clean architecture.

Testing with pytest provides powerful fixtures, parametrization, and plugin ecosystem. Write unit tests for business logic, integration tests for API endpoints, and use mocking for external dependencies.

Async Programming in Python

Asyncio enables concurrent programming using async/await syntax. The event loop manages coroutines, allowing efficient I/O-bound operation handling without threads.

FastAPI leverages async capabilities for high-performance web APIs. It combines Starlette for the web framework, Pydantic for data validation, and automatic OpenAPI documentation generation.

Connection pooling with asyncpg or aiohttp reuses connections efficiently. This reduces the overhead of establishing new connections for each database query or HTTP request.

Task groups and structured concurrency patterns help manage multiple async operations. Python 3.11 introduced TaskGroup for clean exception handling across concurrent tasks.

Async generators yield values asynchronously, useful for streaming data processing. Combined with async for loops, they enable efficient processing of large datasets without loading everything into memory.

Performance Optimization

Profiling identifies bottlenecks before optimization. Use cProfile for CPU profiling, memory_profiler for memory usage, and line_profiler for line-by-line analysis.

List comprehensions are faster than equivalent for loops due to optimized bytecode. Generator expressions save memory by producing values lazily instead of creating entire lists.

Caching with functools.lru_cache or custom implementations avoids redundant computations. For web applications, Redis or Memcached provide distributed caching across multiple instances.

NumPy and Pandas optimize numerical operations through vectorization. Operations on entire arrays are significantly faster than Python loops due to underlying C implementations.

Multiprocessing bypasses the GIL for CPU-bound tasks. ProcessPoolExecutor provides a simple interface, while shared memory enables efficient data sharing between processes.