**What are common applications of Python?**

Python’s versatility makes it suitable for many fields. For example, it’s used in **web development** (frameworks like Django and Flask), **data science and machine learning** (libraries such as NumPy, pandas, scikit-learn, TensorFlow), **scripting and automation** (system administration scripts, task automation), **desktop GUI** (Tkinter, PyQt), **game development** (Pygame), and education due to its readability. In scientific and numeric computing, Python’s powerful libraries make it ideal for research and simulations.

**What are basic Python data types?**

Python’s built-in types include **numeric types** (int, float, complex), **sequence types** (str for text, list, tuple, range), **mapping type** (dict), **set types** (set, frozenset), **boolean** (bool), and **binary types** (bytes, bytearray, memoryview). For example, x = 42 creates an int, s = "hello" is a str, and L = [1,2,3] is a list.

**What is the difference between a list and a tuple?**

A **list** is a mutable sequence (defined with [...]) whose elements can be changed, appended or removed after creation. A **tuple** is an immutable sequence (defined with (...) or no brackets) which cannot be modified once created. In other words, lists are variable-length and changeable, whereas tuples are fixed-size. Because tuples are immutable, they can be used as dictionary keys, whereas lists cannot.

**What is a Python list and how is it different from an array?**

 A Python list is a dynamic, heterogeneous container (elements can be of different types) and can grow or shrink as needed. Python also has an array module and third-party array types (like NumPy’s ndarray) which require fixed data types for performance. In general, use a **list** for general-purpose collections; use specialized arrays (e.g. NumPy arrays) when you need efficient storage and computation on large numeric datasets.

**How do you install external Python packages?**

The standard way is using the **pip** package manager. For example, to install the NumPy library you would run pip install numpy which downloads the package from PyPI and installs it into your environment. You can also specify versions (e.g. pip install package==1.2.3) and upgrade or uninstall packages using pip.

**What is a virtual environment and why use it?**

A virtual environment (created via python -m venv env) is an isolated Python environment with its own interpreter and libraries. It ensures that dependencies for one project don’t conflict with others. Each virtual env has its own site-packages directory, so you can install different versions of packages per project. This helps avoid “dependency hell” and makes projects reproducible.

**What is a Python dictionary?**

A **dictionary** is an *unordered* collection of key:value pairs. You can think of it as a mapping or “address-book” where each unique key maps to a value. For example, d = {"apple": 3, "banana": 5} maps "apple" to 3 etc. Dictionaries provide fast lookup by key and are defined using {} or the dict() constructor. (Keys must be immutable types.)

**How do if, elif, else statements work?** These are Python’s conditional blocks. Example:

if condition1:

# code block A

elif condition2:

# code block B

else:

# code block C

The first true condition’s block executes, and the rest are skipped. Indentation is mandatory to delimit the blocks. Comparison operators (==, <, >, etc.) and logical operators (and, or, not) are used in the condition expressions.

**What loops are available in Python?**

 Python has for loops and while loops. A for loop iterates over a sequence (list, string, tuple, dict keys, etc.), for example:

for x in [1,2,3]:

print(x)

A while loop repeats as long as a condition is true:

i = 0

while i < 5:

print(i)

i += 1

Inside loops, break exits the loop, and continue skips to the next iteration.

**How do you use the range() function in loops?**

The range(start, stop, step) function generates an integer sequence. It’s commonly used with for. For example, range(5) yields 0,1,2,3,4. You can specify a start (default 0) and a step. Example: for i in range(2, 10, 2): iterates over 2, 4, 6, 8. The stop value is exclusive.

**Why is indentation important in Python?**

Unlike languages with braces, Python uses **significant indentation** to define code blocks. All statements within the same block must have the same indentation. Mis-indented code will raise an IndentationError. Proper indentation improves readability, which is part of Python’s design philosophy.

**What is \_\_init\_\_() in a Python class?**

The \_\_init\_\_ method is the class initializer (often called the constructor). It’s automatically invoked when an instance is created. In \_\_init\_\_, you typically assign attributes. For example:

class Dog:

def \_\_init\_\_(self, name):

self.name = name

Here, creating d = Dog("Fido") automatically calls Dog.\_\_init\_\_(d, "Fido"), setting d.name. The \_\_init\_\_ method can take additional parameters for initialization.

**What is self in Python class methods?**

In Python, self refers to the current instance of the class. It is used inside methods to access attributes and other methods on the object. You always include self as the first parameter of instance methods. For example:

class Example:

def \_\_init\_\_(self, value):

self.value = value # 'self.value' is an attribute of this object

def show(self):

print(self.value) # using 'self' to access the attribute

When you call obj = Example(5); obj.show(), Python automatically passes obj as self.

**How does exception handling work?**

Python uses try/except blocks to catch and handle errors. Example:

try:

risky\_operation()

except (TypeError, ValueError) as e:

print("Error:", e)

finally:

print("Cleanup code runs no matter what")

Code in the try block is attempted; if an exception occurs, control jumps to the matching except. You can catch specific exceptions. The optional finally block executes regardless of whether an error occurred. This prevents crashes and allows graceful error handling.

**What is a module and what is a package?**

A **module** is a single Python file (.py) that contains definitions (functions, classes, variables) which you can import into other code. A **package** is a directory containing a special \_\_init\_\_.py file and possibly multiple modules or subpackages. Packages help organize related modules. You import them using import module\_name or from package import submodule.

**What are positional vs keyword arguments in functions?**

In function definitions, parameters can be passed **positionally** or by **keyword**. For example, given def f(x, y=2): ..., you can call f(10) (positional only, so x=10, y=2) or f(10, y=3), or with keywords f(y=3, x=10). Keyword arguments specify the parameter name. You can also define default values (y=2 above). Additionally, \*args collects extra positional arguments into a tuple, and \*\*kwargs collects extra keyword arguments into a dict, allowing flexible argument lists.

**What is a list comprehension?**

A list comprehension is a concise way to create lists using a single expression. Syntax example: [expr(x) for x in iterable if condition(x)]. For instance, [n\*n for n in range(5)] produces [0,1,4,9,16]. It is generally more readable and often faster than an equivalent for loop. Comprehensions can include an optional if to filter items, e.g. [n for n in range(10) if n%2==0] for even numbers.

**What is a generator expression and how is it different from a list comprehension?**

A generator expression is similar to a list comprehension but uses parentheses instead of brackets. For example: gen = (x\*x for x in range(5)). This creates a generator object that **yields** values one at a time, instead of building the entire list in memory. In contrast, a list comprehension would build and return the full list immediately. Generator expressions are memory-efficient for large data. In short, list comprehensions return a full list, while generator expressions return an iterator that produces items on demand.

**What are map(), filter(), and reduce()?**

These are built-in functions (with reduce in functools) from functional programming.

* map(function, iterable) applies the function to each item of the iterable and returns an iterator of results.
* filter(function, iterable) yields only the items for which the function returns True.
* functools.reduce(function, sequence) applies a binary function cumulatively to the items of the sequence, reducing it to a single value (e.g. summing a list). They allow concise data transformations without explicit loops.

**Which testing tools/frameworks are available in Python?**

The standard library provides unittest, a rich framework for unit testing (with test cases and suites). A popular third-party tool is **pytest**, which has a simpler syntax (just write assert statements) and powerful features like fixtures. There’s also nose (less maintained) and doctest (inline tests in docstrings). In practice, many developers use pytest for its ease of use.

**What is \_\_name\_\_ == "\_\_main\_\_" used for?**

In a Python file, the special variable \_\_name\_\_ is set to "\_\_main\_\_" when the file is executed as the main program, and to the module name when imported. So you often see:

if \_\_name\_\_ == "\_\_main\_\_":

main()

This ensures that the code under this if only runs when the script is executed directly, not when it is imported as a module. It’s a common pattern to allow a file to be both used as a script and as an importable module.

**What is dir() and help()?**

dir(object) is a built-in that lists the attributes and methods of an object (module, class, instance, etc.). It’s useful for introspection. help(object) or help("keywords") invokes Python’s help system, showing documentation for the object or topic. These interactive functions help explore code and find available methods or documentation.

**What is slicing in Python?**

Slicing allows you to extract a subsequence from sequences (lists, strings, tuples). The syntax is sequence[start:stop:step]. For example, s = "Hello"; s[1:4] gives "ell" (indices 1 through 3). Leaving out indices means default start/end, and a negative step (e.g. s[::-1]) can reverse the sequence. Slicing is a concise way to manipulate parts of sequences and is very efficient.

**What is the difference between == and is?**

The == operator checks *equality of value*, i.e., whether two objects have the same value. The is operator checks *identity*, i.e., whether two references point to the exact same object in memory. For example, two separate lists with identical contents are == but not is. Generally use == for value comparison and is only to check for the same object (often used for None, e.g. if x is None).

**What are \*args and \*\*kwargs in function definitions?** In a function signature, \*args collects extra positional arguments into a tuple, and \*\*kwargs collects extra keyword arguments into a dict. For example:

def f(a, \*args, \*\*kwargs):

print(a, args, kwargs)

f(1,2,3,x=4,y=5) # a=1, args=(2,3), kwargs={'x':4,'y':5}

This allows functions to accept a variable number of arguments.

**What is a Python package index (PyPI)?**

PyPI is the Python Package Index, the official repository of third-party Python libraries. The pip tool installs packages from PyPI by default. You can publish your own packages to PyPI so others can install them via pip install yourpackage.

**How do you comment code in Python?**

A single-line comment starts with # and continues to the end of the line. For multi-line comments, you can use consecutive # lines or a multi-line string (usually triple quotes """...""") that is not assigned to any variable. (Note: Python does not have a special multi-line comment syntax, only the convention of unused multi-line strings.)

**What is PEP 8?**

PEP 8 is the *Python Enhancement Proposal* that describes the official **Style Guide for Python Code**. It provides conventions on code layout, naming, indentation (4 spaces), maximum line length (79 characters), and more. Adhering to PEP 8 makes Python code more readable and uniform. Many developers use tools like flake8 or black to automatically check or format code to be PEP 8-compliant.

**How do you copy a list in Python?**

You can copy a list in several ways. The safest is using the list’s copy() method or the list() constructor or slicing:

b = a.copy()

b = list(a)

b = a[:] # slice of all elements

These create a shallow copy (the list object is new, but contained elements are references). Don’t use b = a, as that would just create another reference to the same list.

**What is the difference between shallow and deep copy?**

A *shallow copy* of a collection copies the container but not the elements inside; if the elements are mutable objects, both copies refer to the same elements. A *deep copy* recursively copies everything, producing independent elements. In Python, you can perform a deep copy using the copy module’s deepcopy() function.

**How do you open and read a file in Python?**

Use the built-in open() function. For example:

with open("file.txt", "r") as f:

data = f.read()

This opens the file for reading ("r") and ensures it’s properly closed afterwards. You can iterate line-by-line (for line in f:) or read all content at once. To write to a file, use mode "w" or "a".

**What are list, dict, set comprehensions?**

Similar to list comprehensions, Python has comprehensions for other collections:

* **Dict comprehension:** {key\_expr: value\_expr for item in iterable} creates a dictionary.
* **Set comprehension:** {expr for item in iterable} creates a set (unique items). These provide concise ways to build those data structures. For example: {x: x\*x for x in range(5)} yields a dict, and {x for x in some\_list if cond(x)} yields a set of unique x.

**What is the difference between append() and extend() on lists?**

list.append(x) adds a single element x to the end of the list. list.extend(iterable) takes an iterable and appends all its elements to the list. Example:

a = [1,2]

a.append(3) # a is now [1,2,3]

a.extend([4,5]) # a is now [1,2,3,4,5]

If you append([4,5]), it would add the list [4,5] as a single element.

**What is a set and why use it?**

A set is an unordered collection of unique elements. It supports fast membership testing and removes duplicates. For example, s = {1,2,3} or s = set([1,2,2,3]). You can do set operations like union, intersection (&), difference, etc. Use sets when you need uniqueness or fast lookup.

**How do you handle importing modules not in the standard library?**

Use pip install modulename to install it from PyPI into your environment, then import modulename in code. Also ensure your PYTHONPATH or working directory is set correctly if using local modules.

**What is the Python Standard Library?**

It’s a large collection of modules and packages included with Python (no extra install needed). It covers many tasks: file I/O, networking, OS interaction, threading, data formats (JSON, CSV), math, and more. For example, modules like os, sys, math, json, datetime, itertools are in the standard library.

**Explain exception hierarchy in Python.**

All exceptions inherit from the base BaseException class (more commonly catch Exception). Some common built-in exceptions: ValueError, TypeError, IndexError, KeyError, AttributeError, etc. You can catch multiple exceptions (except (TypeError, ValueError):) or a base class to catch many. You can also define custom exceptions by subclassing Exception.

**What is enumerate() and zip()?**

enumerate(iterable) yields pairs (index, item) for each element, useful in loops:

for i, val in enumerate(["a","b","c"]):

print(i, val)

zip(\*iterables) aggregates elements from two or more iterables into tuples:

for x,y in zip([1,2,3], ['a','b','c']):

print(x, y)

It stops at the shortest input. These built-ins simplify many loop patterns.

**What are some common built-in functions?**

Python has many built-ins. Examples include len() (length), type(), isinstance(), open(), range(), min(), max(), sum(), sorted(), enumerate(), zip(), map(), filter(), and more. Knowing these well can save time.

**What is exception chaining?**

You can raise a new exception from an existing one using raise NewException from original\_exception. This preserves the context trace. For example:

try:

...

except KeyError as e:

raise RuntimeError("High-level error") from e

This way the traceback shows both exceptions.

**How do you debug Python code?**

Use print statements or better, use the debugger: the built-in pdb module (import pdb; pdb.set\_trace()). Most IDEs (PyCharm, VSCode) have graphical debuggers with breakpoints. You can also use logging (import logging) at various levels instead of prints, which is more flexible (write to files, adjust severity).

**What are \_\_str\_\_() and \_\_repr\_\_() in classes?**

These are special methods for string representation. \_\_str\_\_(self) should return a “nice” string representation (used by print()). \_\_repr\_\_(self) should return an unambiguous string representation, ideally one that could recreate the object, and is used in the interactive console. If you only define \_\_repr\_\_, it is also used as a fallback for str().

**What is a namespace in Python?**

A namespace is a mapping from names (identifiers) to objects. Each module, function, and class creates its own namespace. For example, global variables in a module are in the module’s namespace, local variables in a function have their own namespace. Python resolves names by checking local, then enclosing, then global, then built-in namespaces (the LEGB rule).

**How does memory management work in Python?**

CPython uses **reference counting**: each object keeps track of how many references point to it, and when the count reaches zero, it is immediately deallocated. In addition, Python has a cyclic garbage collector to detect and collect reference cycles. Memory is managed automatically; as a developer you usually only free memory by removing references or using del. (Be mindful of large data; sometimes using generators or deleting temporary objects can help reduce memory use.)

**What is GIL (Global Interpreter Lock)?**

In CPython, the GIL is a mutex that allows only one thread to execute Python bytecode at a time. This simplifies memory management (single thread modifies object internals), but means multi-threaded Python programs don’t run bytecode in parallel on multiple cores. Only one thread runs at once, making threading less effective for CPU-bound tasks. To leverage multiple cores, you use multiprocessing (separate processes) or asyncio for I/O-bound concurrency.

**What is multiprocessing vs multithreading?**

Python’s threading module runs threads in the same process and memory space. Due to the GIL, only one thread executes Python code at a time, so threads are best for I/O-bound tasks. The multiprocessing module spawns separate processes (each with its own Python interpreter and memory) allowing true parallel execution on multiple CPUs (bypassing the GIL). Use multiprocessing for CPU-bound parallelism. Each process is heavier to create than a thread, but they can fully use multiple cores.

**What is asyncio and when to use it?**

The asyncio library is Python’s standard framework for asynchronous I/O using async/await syntax. It is designed for concurrent I/O-bound and high-level structured network code. An async def function returns a coroutine; you can await other coroutines. asyncio uses an event loop to schedule and run these coroutines. It’s ideal for writing single-threaded concurrent code (e.g. web servers, network clients) without blocking. The official docs summarize: “asyncio is a library to write concurrent code using the async/await syntax”.

**What is type() used for in Python, and how does it help in debugging or writing generic code?**

type() is a built-in function in Python that returns the type of an object. For example, type(3) returns <class 'int'>. It’s commonly used for debugging or when writing functions that need to behave differently based on input types. While Python supports dynamic typing, checking types can help prevent errors and clarify behavior, especially in codebases that use polymorphism or dynamic data sources.

Intermediate/Senior Level Questions

**Explain Object-Oriented Programming (OOP) in Python.**

Python supports OOP: you define classes with the class keyword. A class can have **attributes** (data) and **methods** (functions). You create an instance by calling the class (e.g. obj = MyClass()). Python supports inheritance: a class can derive from one or more base classes, inheriting and optionally overriding their methods. Key concepts: encapsulation (grouping data and methods), inheritance (code reuse), and polymorphism (same method name can operate on different types). Python uses dynamic typing, so there is no need to declare types.

**What are instance methods, class methods, and static methods?**

* **Instance methods** take self as the first parameter and operate on object instances.
* **Class methods** use the @classmethod decorator and take cls as the first parameter (the class itself). They can access/modify class state.
* **Static methods** use @staticmethod and take no special first parameter; they behave like regular functions but live in the class namespace.

Example:

class C:

def f(self): pass # instance method

@classmethod

def g(cls): pass # class method

@staticmethod

def h(x): return x\*x # static method

Use class methods for factory patterns or operations related to the class, static methods for utility functions related to the class.

**What is multiple inheritance and how does Python handle it?**

Python classes can inherit from multiple base classes: class C(A, B). Method resolution order (MRO) defines the order in which base classes are searched for methods. Python uses the C3 linearization algorithm for MRO, ensuring a consistent order. You can examine C.\_\_mro\_\_ or use help(C) to see it. When multiple bases define the same method, the leftmost base in the class definition is used first unless overridden.

**What are magic (dunder) methods?**

These are special methods surrounded by double underscores (e.g. \_\_init\_\_, \_\_str\_\_, \_\_len\_\_, \_\_add\_\_). They allow classes to define or override built-in behaviors (construction, string conversion, arithmetic operators, iteration, etc.). For instance, defining \_\_str\_\_(self) customizes str(obj) output; \_\_eq\_\_(self, other) defines behavior of ==; \_\_enter\_\_/\_\_exit\_\_ enable context managers (with statements). They let objects integrate with Python’s syntax and functions.

**What is a decorator? How do you create one?**

A decorator is a function that takes another function and returns a new function with enhanced behavior. It’s applied using the @decorator syntax above a function. For example:

def my\_deco(func):

def wrapper(\*args, \*\*kwargs):

print("Before")

result = func(\*args, \*\*kwargs)

print("After")

return result

return wrapper

@my\_deco

def greet(name):

print("Hello", name)

greet("Alice")

This will print “Before”, then “Hello Alice”, then “After”. Decorators are often used for logging, authorization checks, caching, etc. Under the hood, @my\_deco transforms greet into greet = my\_deco(greet).

**What is a generator in Python?**

A generator is a special kind of iterator defined by a function using the yield keyword, or by a generator expression. When called, a generator function returns a generator object without running the function body immediately. Each yield pauses the function saving its state, and returns a value to the caller. Calling next() on the generator resumes execution until the next yield. This allows producing values one at a time on-demand. Generators are memory-efficient for large sequences because they don’t build the whole list in memory. For example:

def count\_up\_to(n):

i = 1

while i <= n:

yield i

i += 1

Here, each call to next() yields the next number. Generator expressions provide a quick way to define generators (e.g. (x\*x for x in range(5))). Unlike normal functions that use return, generators use yield, which returns a value and pauses the function (without exiting).

**What are comprehensions and when should you use them?**

Comprehensions (list/dict/set comprehensions) are concise constructs for creating collections from iterables. They often replace loops for building lists, improving readability and sometimes performance. For example, [x\*\*2 for x in nums if x%2==0] quickly creates a list of squares of even numbers. Use them for simple transformations. For more complex logic, traditional loops might be clearer. Generator expressions (x\*\*2 for x in nums) similarly create iterators for lazy evaluation.

**What are modules and packages, and how do you organize code?**

We covered modules and packages earlier. For intermediate, emphasize organization: Place related functionality in modules (e.g. utils.py, models.py). Group related modules into packages (directories with \_\_init\_\_.py). Use absolute or relative imports to access code. Follow a clean project layout, e.g.:

myproject/

app/

\_\_init\_\_.py

models.py

views.py

tests/

requirements.txt

Use if \_\_name\_\_ == "\_\_main\_\_": in scripts to allow modules to be run as programs.

**What are some Python best practices?**

Follow PEP 8 style (naming, indentation, line length), write readable code. Use virtual environments per project. Prefer list/set/dict comprehensions over manual loops when concise. Handle exceptions properly (don’t use bare except:). Write meaningful docstrings and use logging instead of prints for production code. Use immutability (tuples, frozensets) when possible for thread-safety. Keep functions and classes focused (single responsibility).

**What is duck typing in Python?**

Duck typing means Python does not enforce type constraints; if an object implements the required methods or behaviors, it can be used. The name comes from “If it walks like a duck and quacks like a duck”. This allows polymorphism without formal interfaces. For example, any object with a .read() method can be treated as a “file-like object”. Duck typing emphasizes *what an object can do* rather than its class.

**How do you test Python code?**

Use testing frameworks: write **unit tests** that verify small pieces of functionality. With unittest, you subclass TestCase and use assertion methods. With pytest, you write functions starting with test\_ and use plain assert. Run tests automatically via pytest or unittest discovery. Mock external dependencies using the unittest.mock module or libraries like pytest-mock. Aim for high test coverage on critical code.

**What is serialization/pickling in Python?**

Serialization is converting objects to a byte stream. Python’s pickle module can serialize and deserialize Python objects (complex types, custom classes). Example: data = pickle.dumps(obj) and obj = pickle.loads(data). For cross-language or human-readable formats, use JSON (json module) but JSON only handles basic types. Pickle is Python-specific and can be insecure if loading data from untrusted sources.

**What is NumPy and when is it used?**

NumPy is the fundamental library for numerical computing in Python. It provides the ndarray – an N-dimensional, homogeneous array object – and optimized mathematical routines. Use NumPy for efficient array/matrix operations, linear algebra, random sampling, etc. It’s much faster than Python lists for large numeric data, as operations are implemented in C.

**What is Pandas and what is a DataFrame?**

Pandas is a library for data manipulation and analysis. Its primary object is the **DataFrame**, a 2D labeled data structure (like a spreadsheet). A DataFrame stores rows and columns, where each column can have a name and its own data type. You can think of it as a table of heterogeneous data. Pandas provides powerful tools to load data (from CSV, databases), select/filter rows/columns, group and aggregate data, handle missing values, etc. It’s widely used in data science for cleaning and analyzing datasets.

**What is the difference between NumPy arrays and Python lists?**

NumPy arrays (ndarray) have fixed size and homogeneous element type, whereas Python lists can change size and hold different types. NumPy operations (vectorized) are executed in optimized C loops, making them much faster for large numerical data. For example, adding two NumPy arrays adds element-wise in C, whereas adding two lists concatenates them. Use NumPy arrays when performing heavy numerical computations.

**How do you handle missing or null values in pandas?**

In pandas, missing data is represented as NaN or None. Use methods like df.dropna() to remove rows/columns with missing values, or df.fillna(value) to substitute them. Pandas provides functions isnull()/notnull() to detect missingness. Many pandas operations automatically skip NaN (e.g. mean() ignores NaNs). It’s common to fill missing data with statistical values (mean, median) or to drop incomplete rows, depending on context.

**What is Flask and what is it used for?**

Flask is a lightweight WSGI web framework for Python. It is designed to get you started quickly on web projects; you can scale up complexity by adding extensions. Flask provides routing, templating (Jinja2), and a built-in development server. Because it’s minimal, you add only the components you need (ORM, auth, etc). It’s ideal for small to medium web services or APIs.

**What is Django and what is it used for?**

Django is a high-level Python web framework that encourages rapid development and clean design. It is a “batteries-included” framework: it includes an ORM (for database access), an admin interface, authentication, and many utilities out of the box. Django abstracts common web tasks so developers can focus on writing application code. It’s well-suited to larger web applications that benefit from a structured framework.

**When would you choose Flask over Django or vice versa?**

Choose **Flask** when you want a lightweight framework with flexibility to pick components. Flask has a minimal core and is easy to learn, making it great for simple services or microservices. Choose **Django** when you want a full-featured framework with many built-in tools (ORM, admin UI, authentication, etc.) that enforce a standard project structure. Django’s scalability and security features suit large, complex web applications.

**What is an ORM?**

An Object-Relational Mapping (ORM) tool allows you to work with a database using Python classes instead of SQL. For example, Django’s ORM maps Python Model classes to database tables. You can write queries in Python (e.g. User.objects.filter(age\_\_gt=30)) instead of raw SQL. ORMs handle translating operations, managing connections, and provide a higher-level API for CRUD operations. SQLAlchemy and Django ORM are popular Python ORMs.

**What is serialization in Django?**

Django can serialize QuerySets to JSON, XML, etc., using django.core.serializers. This is useful for creating APIs or data interchange. For example: from django.core import serializers; data = serializers.serialize('json', queryset). For REST APIs, many use Django REST Framework which handles serialization with Serializer classes that convert model instances to JSON and vice versa.

**How do you manage dependencies in a Python project?**

Use a virtual environment and keep a requirements.txt listing exact package versions. You can create this with pip freeze > requirements.txt. Others can install the same deps with pip install -r requirements.txt. Alternatively use tools like pipenv or poetry for dependency and virtual environment management, which lock versions and provide deterministic builds.

**What are some best practices for Python code review?**

* Follow PEP 8 style.
* Ensure meaningful variable/function names and docstrings.
* Check for proper exception handling.
* Verify unit tests cover new code.
* Look for potential performance issues (e.g. unnecessary loops).
* Avoid magic numbers; use constants.
* Check for security issues (e.g. input sanitization).
* Use built-in functions and libraries where appropriate (don’t reinvent the wheel).

**What is logging, and how is it different from printing?**

The logging module provides a flexible way to record application events (info, warnings, errors, debug) to files, streams, etc. Unlike print(), logging allows levels, formatting, and easy toggling of verbosity. For example:

import logging

logging.basicConfig(level=logging.INFO)

logging.info("Starting process")

This logs a timestamped message to stderr or a file. In production code, use logging instead of print so you can direct output appropriately and disable verbose logging when needed.

**What are \*args and \*\*kwargs in function calls?** (

Already answered in Q27 with function definitions) in calls, \* unpacks a list/tuple into positional args, and \*\* unpacks a dict into keyword args.) For example:

def f(a, b, c): ...

args = (1,2,3)

f(\*args) # equivalent to f(1,2,3)

d = {'a':1,'b':2,'c':3}

f(\*\*d) # equivalent to f(a=1,b=2,c=3)

**Explain the use of assert.** The assert statement is used for debugging: it tests an expression and raises AssertionError if false. Example: assert len(lst) > 0, "List should not be empty". In optimized runs (python -O), assertions can be skipped, so don’t use them for essential checks, only for sanity checks. Unit tests often use assertions to verify conditions.

**What are lambda functions?**

A *lambda function* is an anonymous (unnamed) function defined with the lambda keyword. It can take any number of arguments but only a single expression. Example: add = lambda x, y: x + y creates a function that returns x+y. Lambdas are often used for short throwaway functions, e.g. as arguments to map(), sorted(), or filter(). They cannot contain statements or annotations – for more complex logic, use a normal def function.

**What is itertools?**

It’s a standard library module that provides tools for efficient looping and combinatorics. It includes functions like count(), cycle(), permutations(), combinations(), product(), and groupby(). These create iterators for infinite sequences or combinatorial sets, saving memory. For example, itertools.groupby() can group sorted data by keys, and itertools.chain() can concatenate multiple iterators.

**What is functools and give examples?**

The functools module contains higher-order functions and utilities. Examples include:

* functools.reduce() for reduction (as above).
* functools.partial(func, \*args) to fix some arguments of a function and generate a new callable.
* @functools.lru\_cache decorator to cache function results.
* functools.wraps to preserve metadata when writing decorators.

**What is the purpose of \_\_slots\_\_ in classes?**

By default, Python objects store attributes in a dynamic dict, which uses more memory per instance. Defining \_\_slots\_\_ = ['attr1','attr2'] in a class restricts it to fixed attributes and omits the instance dict, saving memory and preventing new attributes from being added dynamically. This is an optimization for classes when many instances are created.

**How does slicing work on lists/strings?**

(Already answered partly in Q25, but here mention negative indexing specifically.) Example: lst[-1] is last element, lst[-2:] is the last two elements. s[::-1] reverses a string. Slicing in general is [start:stop:step], where start defaults to 0, stop to end, and negative indices count from the end.

**How do you merge two dictionaries?**

In Python 3.9+, you can use dict3 = dict1 | dict2 (dictionary union operator). In older versions, you can do dict3 = {\*\*dict1, \*\*dict2} or use dict.update(). Note that if there are overlapping keys, the second dictionary’s values overwrite the first’s.

**What is the use of enumerate()?** We covered enumerate in Q41, but emphasize for intermediate: it yields (index, element) pairs. Often used in loops when you need a counter:

for idx, val in enumerate(some\_list, start=1):

print(idx, val)

The optional start parameter can set the initial index (default 0).

**Explain the concept of property in classes.**

The built-in property() function (or decorator @property) allows you to define getter/setter behavior in a class attribute. It lets you access a method like an attribute. For example:

class C:

@property

def x(self):

return self.\_x

@x.setter

def x(self, value):

self.\_x = value

Now obj.x returns self.\_x and setting obj.x = 5 calls the setter. This provides controlled attribute access without changing the public API.

**What is multi-threading? Does Python support it?**

Multi-threading means running threads (lightweight sub-processes) in a program. Python supports threading via the threading module. However, due to the GIL, CPU-bound threads don’t run in parallel. Threads are useful for I/O-bound concurrency (e.g. waiting for network). Example:

import threading

def task():

print("Hello from thread")

t = threading.Thread(target=task)

t.start()

t.join()

This runs task in a separate thread.

**What is concurrency and parallelism in Python?**

Concurrency means handling multiple tasks at overlapping times (e.g. using async IO or threads), while parallelism means executing multiple tasks simultaneously on multiple CPU cores (typically via multiprocessing). In Python:

* **Concurrency** can be achieved with asyncio or threads (for I/O-bound tasks).
* **Parallelism** is achieved with multiprocessing or concurrent.futures.ProcessPoolExecutor, which run code on different CPUs.

**What are design patterns in Python (e.g. Singleton, Factory)?**

Python can implement common design patterns. For example, a Singleton (only one instance) can be done by using a class with a class variable holding the instance, or by using a metaclass. A simple example using a module-level instance or overriding \_\_new\_\_. Patterns often look simpler in Python due to its dynamic nature. Discussing specific patterns often depends on the job context.

**What is a context manager? Give an example.**

A context manager is an object that defines \_\_enter\_\_ and \_\_exit\_\_ methods to set up and tear down resources. The with statement uses context managers. Example:

with open('file.txt','r') as f:

data = f.read()

Here, open() returns a context manager whose \_\_enter\_\_ returns the file handle, and its \_\_exit\_\_ ensures the file is closed. You can create your own by defining a class with \_\_enter\_\_ and \_\_exit\_\_, or by using the contextlib utilities.

**How do you run asynchronous code?**

Use async def to define coroutine functions, and await to call other async functions. You typically create an event loop:

import asyncio

async def hello():

print("Hello")

await asyncio.sleep(1)

print("World")

asyncio.run(hello())

Here, asyncio.run() manages the event loop. In frameworks (like FastAPI or aiohttp), the loop is managed for you.

**What are *coroutines*?**

In Python’s asyncio, a coroutine is a function defined with async def. Calling it returns a coroutine object. You use await inside coroutines to wait on other coroutines or asynchronous operations. Coroutines allow writing asynchronous code in a sequential style. They must be run in an event loop or awaited.

**What is metaprogramming?**

Metaprogramming is writing code that manipulates code (classes, functions) at runtime. In Python, metaprogramming examples include using decorators, introspection (getattr, setattr), dynamic class creation (via type()), or metaclasses. It’s an advanced technique to make flexible libraries (ORMs use this). For example, dynamically creating classes or adding methods at runtime is metaprogramming.

**What is memory leak in Python and how do you prevent it?**

Python’s garbage collector handles most memory. However, memory leaks can occur if objects are kept alive unintentionally (e.g. circular references in objects with \_\_del\_\_, or caching large data). To prevent leaks, remove references when done (e.g. del large structures or remove items from caches), and avoid unnecessary circular references. The gc module can help debug leaks. Using weak references (weakref) for caches can also prevent leaks.

**How do you profile and optimize Python code?**

Use the cProfile or profile module to collect execution statistics. For example:

import cProfile

cProfile.run('my\_function()')

This shows time spent per function call. You can then optimize hot spots (e.g. replace slow Python loops with list comprehensions, use built-in functions, or move critical code to C extensions if needed). The timeit module can measure small code snippets. Also consider alternative implementations like PyPy or use numpy for numeric loops.

**What is a Python C extension?**

Python can be extended with modules written in C for performance or to interface with C libraries. A C extension module is compiled and imported like a regular Python module. You use the CPython API to create extension types and functions. Tools like Cython or cffi ease this process by generating the C glue code. C extensions can greatly speed up compute-intensive tasks by moving them to native code.

**What are generators and iterators?**

We covered generators in Q56. An **iterator** is any object with \_\_iter\_\_() and \_\_next\_\_() methods. It represents a stream of data that can be iterated over. Lists and tuples are iterable but not iterators (they create an iterator via iter()). Generators are a convenient way to create iterators. Example of a custom iterator:

class Countdown:

def \_\_init\_\_(self,n): self.n = n

def \_\_iter\_\_(self): return self

def \_\_next\_\_(self):

if self.n <= 0: raise StopIteration

self.n -= 1

return self.n+1

**What is monkey patching?**

Monkey patching means dynamically modifying or extending code at runtime, such as adding attributes or methods to existing classes or modules. For example, you could assign str.is\_palindrome = lambda self: self == self[::-1]. While powerful for quick fixes or testing, monkey patching is risky (can break future updates) and should be used judiciously (often in testing to mock behavior).

**How are exceptions propagated?**

If an exception is not caught in a function, it propagates (bubbles up) to the caller. If still unhandled, it goes up the call stack until it reaches the top level, where it terminates the program and prints a traceback. You can catch exceptions at any level with try/except, and use raise to re-raise or raise NewException from e to chain exceptions. Properly catching exceptions prevents propagation.

**What is Virtualenv vs venv?**

venv is the standard module (since Python 3.3) for creating lightweight virtual environments. virtualenv is an older third-party tool that also creates isolated environments. Both create folders with separate Python executables and site-packages. Today, python -m venv env is the common approach.

**How does Python’s contextlib help manage resources? Provide an example using contextlib.contextmanager.**

The contextlib module provides utilities for working with context managers—objects used with with blocks to manage resources safely. The @contextmanager decorator allows writing a generator-based context manager without creating a full class. For example:

from contextlib import contextmanager

@contextmanager

def open\_file(path, mode):

f = open(path, mode)

try:

yield f

finally:

f.close()

Using this, with open\_file("log.txt", "w") as f: automatically handles file closing, even if an error occurs. This approach simplifies resource management for custom logic, making your code cleaner and more robust.

**What is containerization and how do you Dockerize a Python application?**

Containerization means packaging an application and all its dependencies into a portable image. To Dockerize a Python app, you typically write a Dockerfile starting from an official Python base image. For example, you might begin with FROM python:3.11-slim, which provides a lightweight Python runtime. In the Dockerfile you set up environment variables (such as PYTHONDONTWRITEBYTECODE=1 for performance), create a working directory, and copy your code into it. You then install dependencies with pip (e.g. RUN pip install -r requirements.txt), and finally specify a command to run the app (using CMD or ENTRYPOINT). This results in a container image that encapsulates Python, your code, and all libraries, ensuring the app runs the same way on any machine.

Advanced/Master Level Questions

**What is the Global Interpreter Lock (GIL) and how does it affect**

**concurrency?**

The GIL is a mutex in CPython that allows only one thread to execute Python bytecode at a time. This means even in a multi-threaded Python program, threads don’t run Python code in parallel on multiple cores – only one executes at any moment. The GIL simplifies memory management (since only one thread touches Python objects) but is a bottleneck for CPU-bound multi-threading. The impact is minimal for single-threaded or I/O-bound code, but limits pure-Python multi-threaded throughput.

**How can you achieve parallelism in Python?**

Use the multiprocessing module or concurrent.futures.ProcessPoolExecutor to spawn separate processes (each with its own interpreter and memory) allowing true parallel execution. This bypasses the GIL at the cost of higher overhead. Alternatively, use implementations without a GIL (Jython, IronPython) or use C extensions/Numba for heavy computation.

**Explain Python’s memory model and garbage collection.**

CPython uses reference counting for memory: each object has a reference count that increases/decreases as variables refer to it. When the count hits zero, the object is deallocated immediately. For cyclic references (where ref count doesn’t drop to zero), Python also has a cyclic garbage collector (in gc module) that periodically detects and frees such cycles. Developers can invoke gc.collect() manually if needed. Understanding this helps manage memory, e.g. removing circular refs or clearing large data.

**What are *design patterns* (e.g., Singleton, Factory) in Python?**

Design patterns are typical solutions to common problems. For example, a **Singleton** ensures only one instance of a class: in Python you can do this with a module (modules are singleton by nature) or by controlling instance creation in \_\_new\_\_. A **Factory** pattern provides an interface for creating objects (a function/class method that returns different classes based on parameters). Python’s dynamic features (first-class functions, classes) often allow simpler pattern implementations than in static languages.

**What are coroutines and async/await in depth?**

Coroutines (in asyncio) are more than generators: defined with async def, they use await to yield control. When you await an async function, it suspends until the awaited task is done, allowing other coroutines to run in the meantime. The event loop schedules coroutines and I/O events. Unlike callbacks, async/await leads to more readable sequential-style code. Internally, coroutines throw a StopIteration with the return value when finished. You can gather multiple coroutines with asyncio.gather().

**How do you use the asyncio event loop?**

The asyncio event loop drives execution of coroutines and callbacks. High-level, use asyncio.run(main()) to start your async code. Within async functions, use await. For low-level control, you can get the loop with loop = asyncio.get\_event\_loop(), schedule tasks with loop.create\_task(), and run until complete with loop.run\_until\_complete(). Callbacks can be scheduled with loop.call\_soon(). Understanding the loop is key for integrating async code with other systems.

**What are descriptors in Python?**

Descriptors are objects that define how attribute access works. An object implements \_\_get\_\_, \_\_set\_\_, or \_\_delete\_\_, it’s a descriptor. For example, methods are descriptors (functions with \_\_get\_\_ to bind self). The @property decorator uses descriptors under the hood. Descriptors allow customization of attribute access (logging, validation). They are a more advanced way to manage attributes beyond \_\_getattr\_\_.

**Explain metaclasses and give an example use.**

A metaclass is the “type” of a class: it’s a class that creates classes. By default, Python uses type as the metaclass. You can create a custom metaclass by subclassing type and override \_\_new\_\_ or \_\_init\_\_ to customize class creation. For example, a metaclass could automatically register all classes that use it, or inject new methods into a class at creation time. In essence, “a metaclass functions as a template for classes”. Use cases include enforcing interface contracts or automatically adding attributes.

**What is type hinting and how is it used?**

Python 3.5+ supports optional type hints (PEP 484). You can annotate function signatures and variable declarations for documentation and static analysis. Example: def f(x: int, y: str) -> bool: hints that x should be an int, y a str, and the return type is bool. The typing module provides complex hint types (List[int], Optional[str], etc.). Python does not enforce types at runtime, but tools like mypy or IDEs can check consistency. Type hints improve code clarity and help catch errors.

**What are some memory optimization techniques in Python?**

For large data, prefer generators over lists, use \_\_slots\_\_ to reduce instance memory, use built-in functions (they’re in C). Release large objects when done (e.g. del or reassign). Avoid circular references if unnecessary. In numeric contexts, use NumPy arrays instead of Python lists to save memory. When working with strings, prefer join() rather than concatenation in loops. Profile memory usage with tools like memory\_profiler if needed.

**How do you perform performance profiling on Python code?**

Use the cProfile module to see where time is spent. Example:

import cProfile, pstats

cProfile.run('my\_func()', 'profile.stats')

p = pstats.Stats('profile.stats')

p.sort\_stats('cumtime').print\_stats(10)

This lists the most time-consuming functions. Also timeit is useful for small snippets. For memory profiling, use tracemalloc (built-in) or external tools like memory\_profiler. After identifying bottlenecks, optimize code (e.g. using more efficient algorithms or data structures, caching results, or writing critical parts in C/Cython).

**How do you manage concurrency issues like race conditions?**

Use synchronization primitives from the threading module: Lock, RLock, Semaphore, Event, etc. For example, a Lock ensures only one thread accesses a shared resource at a time. Alternatively, use higher-level constructs like queue.Queue for thread-safe queues, or use multiprocessing which avoids shared state. In async code, avoid shared state or use asyncio.Lock. Prefer immutable data or patterns that minimize shared mutability.

**What is asyncio’s gather() and wait()?**

 asyncio.gather(\*coros) runs multiple coroutines concurrently and returns a future that yields all results (in order). It raises exceptions if any coroutine fails. asyncio.wait() returns two sets of tasks (done, pending) and can be used for more complex concurrency control. These are ways to run multiple coroutines at once.

**What are properties of Python’s I/O (file/network) operations?**

Python’s I/O (file, socket) is generally blocking by default. You can use non-blocking or asynchronous I/O by using threads, asyncio, or libraries like aiofiles. The file API is buffered, so file.read() may block until data is available or EOF. Network libraries (like socket) can be set to non-blocking mode or integrated with select/asyncio. Understanding blocking behavior is important for performance and concurrency.

**How do you deal with large files or data streams?**

Process them incrementally. For example, read large files line by line (for line in file:) instead of read() all at once. Use generators to stream data. For CSV or JSON, use iterative parsing (e.g. pandas.read\_csv(chunksize=...) or ijson for streaming JSON). This avoids loading entire files into memory.

**What is Cython?**

Cython is a tool to compile Python-like code (with optional static type annotations) into C for performance. You write .pyx files which can include type declarations, and Cython generates C code which is compiled into a Python extension module. This can greatly speed up loops and math by moving them into compiled code. Cython is often used for optimizing hotspots in Python applications.

**What is a wheel (.whl) and a wheel file?**

A wheel is a built-package format for Python (PEP 427). It’s a ZIP archive with a standardized name (including Python and platform tags) that can be installed with pip. Creating wheels allows distributing compiled extensions for faster install. When you pip install, pip often prefers wheels if available (because building from source can take time).

**What are some advanced features in Python 3.x?** Examples include:

* **async/await** (as above) for asynchronous programming.
* **Type hinting** (PEP 484) for optional static typing (we covered).
* **f-strings** for formatted string literals (introduced in 3.6) e.g. f"Hello {name}".
* **Data classes** (@dataclass in 3.7) for boilerplate-free classes.
* **Pathlib** for object-oriented filesystem paths.
* **Enum** support (enum module) for enumerations.
* **Keyword-only arguments** (functions can enforce keyword args using \* in signature).
* **Assignment expressions** (the “walrus operator” := in 3.8).

**What are dataclasses?**

The dataclasses module (Python 3.7+) provides a decorator to automatically generate special methods in classes (like \_\_init\_\_, \_\_repr\_\_, and others) for classes that are mostly data containers. Example:

from dataclasses import dataclass

@dataclass

class Point:

x: float

y: float

This automatically creates an \_\_init\_\_, so you can do Point(1,2), and a \_\_repr\_\_ like Point(x=1, y=2). It saves boilerplate and supports features like default values and comparison.

**What is introspection and how is it useful?**

Introspection is examining objects at runtime (e.g. using type(), dir(), hasattr(), inspect module). It allows dynamic code behavior. For example, a library might inspect a class’s methods to dynamically bind them, or to auto-generate documentation. Python’s ability to inspect its own objects enables many frameworks (ORMs, serializers, test frameworks) to work without explicit configuration.

**What is pickling/unpickling and its security implications?**

(We touched on serialization.) Using pickle, you serialize objects to bytes and pickle.loads() to recreate them. However, unpickling data from untrusted sources can execute arbitrary code, making it a security risk. Only unpickle data you trust. For safer data exchange, prefer JSON or other formats where the structure is explicit.

**What are some ways to handle missing dependencies at runtime?**

You can use exception handling around imports:

try:

import optional\_lib

except ImportError:

optional\_lib = None

Or specify dependencies in setup.py. You can also check sys.version\_info for version-specific code. For optional features, gracefully disable functionality if a module is missing.

**What is the difference between a module’s global variable and a class’s static variable?**

A module global is a variable at module level, shared across all uses of that module. A class’s static variable (class attribute) is shared by all instances of that class. Both act as shared state: for example, MyClass.count = 0 is shared by every MyClass instance. The difference is organizational (module vs class scope), but behavior is similar: modifying it affects all references.

**Describe Python’s execution model for function calls.**

When you call a function, Python pushes a new frame onto the call stack, with its own local namespace. The function arguments are evaluated before the call, and if they are mutable, changes can affect the caller’s objects. After execution, the frame is popped. This model supports recursion naturally. Tail-call optimization is not performed in CPython (so deep recursion can lead to RecursionError).

**Explain asynchronous generators and comprehensions.**

(Advanced) Python 3.6 introduced async generators: async def agen(): yield .... You can async for over them. There are also asynchronous comprehensions: [x async for x in aiter]. These allow similar patterns in async code. For example:

async def async\_range(n):

for i in range(n):

yield i

async def main():

async for x in async\_range(5):

print(x)

This is niche but useful in async-heavy code.

**What is multi-processing and what are the new features (e.g. spawn/fork)?**

The multiprocessing module lets you use processes. On Unix, the default start method was “fork” (child process is a copy). On Windows (and optionally on Unix), there’s “spawn” (start fresh Python interpreter) and “forkserver”. You can choose with multiprocessing.set\_start\_method(). Fork can be faster but may have issues with threads (inheriting state), so “spawn” is safer and default on Windows.

**What are some strategies for scaling Python web applications?**

Scale by running multiple worker processes (via WSGI servers like Gunicorn with multiple workers), use load balancers, or container orchestration (Kubernetes). Offload static files to a CDN. Use caching (Redis, Memcached). For CPU-bound tasks, run them in background workers (e.g. Celery with multiple processes). Use asynchronous frameworks (FastAPI, aiohttp) to handle many concurrent I/O-bound requests. Profile and optimize bottlenecks, and consider sharding or microservices for large loads.

**What is the difference between threading and asyncio?**

Threading uses OS threads to allow concurrency (though limited by GIL), suitable for I/O-bound tasks. asyncio uses a single-threaded event loop and coroutines (async/await) to handle concurrency; it doesn’t use threads by default. Asyncio is often more efficient (no thread overhead) for very many small I/O tasks, but code must be written in async style. Threading can use libraries that block, whereas async code requires non-blocking libraries.

**What are Python’s built-in optimization techniques?**

Python applies a few optimizations automatically: short-circuit evaluation for boolean ops, caching small integers/strings, and compiled bytecode caching (.pyc files). You can manually optimize by using built-ins (sum(), list comprehensions), avoiding global lookups (local variables are faster), and minimizing attribute lookups (use local variables for frequently accessed attributes).

**What is a memory view?**

A memoryview is a built-in that allows Python code to access the memory of binary objects (like bytes, bytearray) without copying. It supports slicing and views large data buffers. Useful for zero-copy manipulation of binary data, e.g. in networking or image processing. Example:

ba = bytearray(b"Hello")

mv = memoryview(ba)

mv[0] = 74 # changes ba to b'Jello'

**How do you ensure code quality in large Python projects?**

Use linters (flake8, pylint), formatters (black), type checkers (mypy) for consistency. Enforce code style with CI tools. Write comprehensive unit/integration tests and use continuous integration to run them on every commit. Perform regular code reviews. Use dependency management (requirements.txt with exact versions or lock files) and documentation (docstrings, Markdown docs).

**How do you call C code from Python?**

Several ways:

These allow leveraging existing C libraries or writing performance-critical code in C.

* **ctypes**: a stdlib library to call C functions in shared libraries (DLLs) at runtime without compiling Python extension.
* **cffi**: a library to call C code by interfacing with the C ABI.
* **Cython or Python C API**: write C extension modules (compile .pyx or .c files).
* **subprocess**: run a separate C program as a process.

**What is the PyPy project?**

PyPy is an alternative Python interpreter with a JIT compiler, often faster for long-running code. It is mostly compatible with CPython but may have issues with C extensions (though support has improved). PyPy can run many Python programs faster without code changes. Mention as an option for performance tuning.

**What tools can be used for static code analysis?**

Tools include mypy for type checking (if you use type hints), pylint or flake8 for style and linting, bandit for security checks, and safety or pip-audit for checking known vulnerabilities in dependencies. These tools help catch errors before runtime.

**What is the purpose of Python Enhancement Proposals (PEPs)?**

PEPs are design documents for the Python community. They propose new features (like PEP 8, PEP 484 for type hints). Reading relevant PEPs (they are public on peps.python.org) gives insight into language decisions and features.

**Explain Python’s import system and \_\_init\_\_.py.**

When you import module, Python searches the module in its import path (directories in sys.path). A directory is recognized as a package if it contains an \_\_init\_\_.py (even empty). In Python 3.3+, namespace packages allow packages without \_\_init\_\_.py. The import mechanism caches loaded modules in sys.modules. Use absolute imports for clarity, and relative imports (from . import sibling) for intra-package references.

**What are PEP 257 and docstrings?**

PEP 257 defines conventions for Python docstrings. Each module, class, function should have a docstring right under the definition using triple quotes. Docstrings should describe the object’s purpose and usage. Tools like Sphinx or pydoc can auto-generate documentation from docstrings. Follow PEP 257 for multi-line docstring formatting.

**What are some advanced debugging techniques?**

Use pdb for step-by-step debugging, or higher-level tools like ipdb (IPython-enabled) or IDE debuggers. Use breakpoints (import pdb; pdb.set\_trace()). Inspect variables, stack, and execution flow. For multi-threaded programs, use thread-aware debuggers. Profilers (cProfile, line\_profiler) help locate performance issues. For memory leaks, gc module and tracers can help.

**What are Python interpreters other than CPython?**

Examples: PyPy (JIT-compiled), Jython (for Java), IronPython (for .NET), MicroPython (for microcontrollers). Each implements Python in different environments or with performance trade-offs. CPython is the standard C-based implementation. Knowing alternatives can be useful for specific use cases.

**What are design principles like EAFP vs LBYL?**

Python often follows the EAFP (Easier to Ask Forgiveness than Permission) principle: try an operation and catch exceptions, rather than checking pre-conditions. This is opposite of LBYL (Look Before You Leap). For example, instead of checking if key in dict: val = dict[key], one might do try: val = dict[key] and catch KeyError. EAFP idiom is considered Pythonic.

**What are context variables (PEP 567)?**

Python 3.7+ introduced context variables for managing context in async tasks (like thread-local storage but for async). Use contextvars to store data that is local to a coroutine. They allow separating contexts in concurrent code.

**How do you achieve scalability in Python for big data/ML applications?**

Use distributed processing frameworks (Spark, Dask) or cloud services. For web ML services, use microservices, GPU acceleration (TensorFlow/PyTorch with CUDA), and horizontal scaling of API servers. Choose appropriate data structures (NumPy arrays, sparse matrices). Offload heavy work to optimized libraries or external systems. Caching (Redis), message queues (RabbitMQ, Kafka), and load testing all help ensure scalability.

**What is OpenTelemetry and how do you use it for observability in Python?**

OpenTelemetry is an open-source observability framework for generating and collecting telemetry data (metrics, logs, traces) from applications. In Python, you install the OpenTelemetry API and SDK packages with pip (for example, pip install opentelemetry-api opentelemetry-sdk). You can then instrument your code – either manually or via auto-instrumentation libraries – to emit spans, metrics, and logs. This data is sent to backends (e.g. Jaeger, Prometheus, or an OTLP endpoint) for analysis. Using OpenTelemetry lets a Python service produce structured telemetry in a standardized way, which greatly aids debugging and performance monitoring across distributed systems.

**Explain Python’s structural pattern matching (match-case) introduced in Python 3.10. Provide an example.**

Structural pattern matching (PEP 634) adds a match statement that compares a value against a series of patterns. The syntax is:

match subject:

case <pattern1>:

<action1>

case <pattern2>:

<action2>

case \_:

<fallback>

The subject is evaluated and checked against each case in order, executing the first matching block. Patterns can be literals, types, sequences, or even class instances. For example:

def http\_error(status):

match status:

case 400:

return "Bad request"

case 404:

return "Not found"

case \_:

return "Something else"

Here, if status is 404, the function returns "Not found", and if no literal matches, the wildcard \_ case handles the rest. Pattern matching is more powerful than simple if/elif chains when you need to destructure data or match on complex conditions.

**What are Exception Groups and the except\* syntax introduced in Python 3.11 (PEP 654)? When would you use them?**

PEP 654 (Python 3.11) introduced **Exception Groups**, which allow a single raise to combine multiple exceptions into one object. The new except\* syntax lets you catch and handle specific exception types within an exception group. This is useful in concurrency: for example, if several coroutines fail simultaneously, asyncio can raise an ExceptionGroup containing all errors. You can then write:

try:

await asyncio.gather(task1(), task2())

except\* (ValueError, TypeError) as eg:

# handle ValueError and TypeError from any task

In this way, except\* matches exceptions of the given type inside the group. It provides finer-grained error handling when multiple errors occur together.

**What is LangChain, and how is it used in Python for LLM-based**

**applications?**

LangChain is a Python framework for building applications powered by large language models (LLMs). It provides standard interfaces for language models, prompting, memory, and data connectors, and integrates with many LLM providers. LangChain simplifies common patterns in LLM workflows: for example, an LLMChain allows you to pair a prompt template with an LLM call, and components like RetrievalQA combine embeddings and vector stores to answer questions from documents. By using LangChain, developers can compose chains or agents of LLM calls and other tools in a structured way, making it easier to develop, test, and deploy generative AI applications.

**What is FastAPI and what advantages does it offer for modern Python web APIs?**

FastAPI is a modern, high-performance Python web framework for building APIs. It is based on standard Python type hints (using Pydantic for data validation), which enables **automatic request validation** and **interactive API docs**. Because FastAPI builds on Starlette and uses async features, it can achieve performance comparable to Node.js or Go. For example, by declaring a request model with types, FastAPI automatically generates OpenAPI (Swagger) documentation for your endpoints. This type-driven design means faster development (with strong editor support) and reliable APIs. Overall, FastAPI’s use of standard Python features and async support makes it both developer-friendly and production-ready.

**How does Python’s ParamSpec and TypeVarTuple enhance function typing, and when would you use them?**

Python’s typing system has evolved to support more dynamic and flexible function signatures. These enhancements provide greater flexibility and precision in type annotations, facilitating better code clarity and tooling support:

* **ParamSpec**: Introduced in PEP 612, ParamSpec allows for precise typing of callable parameters, especially useful when writing decorators that need to preserve the signature of the functions they wrap.
* from typing import Callable, ParamSpec, TypeVar
* P = ParamSpec('P')
* R = TypeVar('R')
* def decorator(func: Callable[P, R]) -> Callable[P, R]:
* def wrapper(\*args: P.args, \*\*kwargs: P.kwargs) -> R:
* # Pre-processing
* result = func(\*args, \*\*kwargs)
* # Post-processing
* return result
* return wrapper
* **TypeVarTuple**: Proposed in PEP 646, TypeVarTuple allows for variadic generics, enabling functions and classes to accept an arbitrary number of type parameters. This is particularly beneficial when working with functions that accept a variable number of arguments, such as in mathematical computations or data processing pipelines.

**What are the implications of Python 3.13’s free-threaded mode on C extensions?**

Python 3.13 introduces an experimental **free-threaded** mode, aiming to remove the Global Interpreter Lock (GIL) and allow true multi-threaded execution. This change has significant implications for C extensions:

* **ABI Incompatibility**: C extensions compiled for the traditional GIL-based Python interpreter are incompatible with the free-threaded version due to differences in the Application Binary Interface (ABI). Developers must ensure their extensions are compiled specifically for the targeted Python build.
* **Thread Safety**: Extensions must be audited and potentially refactored to be thread-safe, as the absence of the GIL means that concurrent access to shared resources must be managed explicitly to avoid race conditions.
* **Distribution Complexity**: Package maintainers may need to distribute multiple wheels to support both GIL and free-threaded versions, increasing the complexity of package management.

**How can you profile and address numerical instability in Python applications?**

Numerical instability can lead to significant errors in scientific and engineering computations. By proactively profiling and addressing numerical instability, developers can enhance the reliability and accuracy of their Python applications, particularly in domains requiring high-precision computations. To profile and mitigate such issues in Python:

* **Use Specialized Profilers**: Tools like **PyTracer** can automatically instrument Python code to detect and quantify numerical instabilities. PyTracer tracks numerical operations and identifies areas where rounding errors or loss of significance may occur.
* **Implement Monte Carlo Arithmetic**: Introducing controlled random perturbations in computations can help assess the sensitivity of algorithms to numerical errors.
* **Adopt Arbitrary Precision Libraries**: For critical computations, libraries like decimal or mpmath provide arbitrary precision arithmetic, reducing the risk of floating-point errors.
* **Refactor Algorithms**: Analyze and refactor algorithms to improve numerical stability, such as by avoiding subtraction of nearly equal numbers or by reordering operations to minimize error propagation.