Python Interview Questions & Answers 2026

Random Set • March 24, 2026

Q1. Explain 'Efficiently Combining, Counting, and iterating' in detail. Why is it important in 2026?

Efficiently combining, counting, and iterating over data is a core skill in Python — it lets you merge sequences, tally frequencies, and process items with minimal memory and maximum speed. In 2026, with large datasets, streaming inputs, and performance-critical code, using the right tools — zip() for parallel combining, collections.Counter for fast counting, and generators/comprehensions for memory-safe iteration — turns slow loops into clean, scalable operations that run in production without crashing or slowing down. Here’s a complete, practical guide to efficiently combining, counting, and iterating: core techniques, real-world patterns, performance wins, and modern best practices with type hints, la...

Category: Efficient Code • From: Efficiently Combining, Counting, and iterating

Q2. How does CrewAI vs LangGraph vs AutoGen 2026 – Which Framework Should You Use? work? Give a practical example.

In 2026, building Agentic AI systems with Python has become mainstream. Three frameworks dominate the landscape: CrewAI , LangGraph (from LangChain), and AutoGen (from Microsoft). Each has different strengths, making the choice dependent on your specific use case. This detailed 2026 comparison will help you decide which Agentic AI framework is best for your project. Quick Comparison Table – CrewAI vs LangGraph vs AutoGen (March 2026) Aspect CrewAI LangGraph AutoGen Best For Role-based multi-agent teams Complex stateful workflows Conversational multi-agent systems Ease of Use Very Easy (most beginner-friendly) Intermediate to Advanced Intermediate Control & Flexibility Good E...

Category: Agentic AI • From: CrewAI vs LangGraph vs AutoGen 2026 – Which Framework Should You Use?

Q3. Explain 'Index lookups' in detail. Why is it important in 2026?

Index lookups are the foundation of accessing elements in strings, lists, tuples, and other sequences in Python — using square brackets [] with integer positions to retrieve individual items or slices. Positive indices start at 0 (first element) and go up to len(sequence) - 1 (last element); negative indices count backward from -1 (last element) to -len(sequence) (first element). Indexing returns a single element; slicing with [start:end:step] returns a new sequence (substring for strings, sublist for lists). In 2026, index lookups remain essential — used constantly in text parsing, data extraction, log analysis, pandas/Polars column operations, array manipulation, and any task requiring positional acc...

Category: Regular Expressions • From: Index lookups

Q4. Explain 'Filtering a chunk' in detail. Why is it important in 2026?

Filtering a chunk when reading large CSV files with pd.read_csv(chunksize=...) is a key technique for memory efficiency — apply filters, transformations, or aggregations to each chunk immediately, discard unnecessary rows early, and only keep (or append) the processed results. This drastically reduces peak memory usage, speeds up processing, and avoids loading irrelevant data into the final DataFrame. In 2026, chunk filtering is standard practice for big data ETL, cleaning, feature engineering, or subsetting — especially when files are gigabytes large, RAM is limited, or you're only interested in specific rows (e.g., recent dates, high-value records, certain categories). It pairs perfectly with pandas query...

Category: Parallel Programming With Dask • From: Filtering a chunk

Q5. What are the best practices for UTC offsets in modern Python development?

UTC offsets are the standard way to represent the difference between local time and Coordinated Universal Time (UTC) — expressed as hours and minutes ahead (+HH:MM) or behind (-HH:MM) UTC, with Z or +00:00 for UTC itself. In Python, you attach offsets to datetime objects using tzinfo — either fixed offsets via timezone(timedelta(...)) or named time zones via zoneinfo.ZoneInfo (Python 3.9+ built-in) or pytz . In 2026, UTC offsets are non-negotiable in production code — they prevent daylight saving bugs, ensure correct time math across regions, and make timestamps interoperable in logs, APIs, databases, and distributed systems. Here’s a complete, practical guide to UTC offsets in Python: fixed...

Category: Dates and Time • From: UTC offsets

Q6. How does Default and flexible arguments work? Give a practical example.

Default and flexible arguments make Python functions incredibly versatile — letting callers provide only what they need while still supporting rich, customizable behavior. Default parameters give optional inputs sensible fallbacks, while flexible arguments (*args for positional, **kwargs for keyword) allow any number of extra values. Together, they create clean, adaptable APIs that feel natural to use. In 2026, modern Python functions combine defaults, *args, **kwargs, type hints, and clear docstrings to handle everything from simple calls to complex configurations. Here’s a complete, practical guide to using them effectively — with real patterns and best practices. Start with defaults: they make param...

Category: Data Science Tool Box • From: Default and flexible arguments

Q7. How does Reordering values work? Give a practical example.

Reordering values in string formatting lets you control the exact order in which variables appear in the output string — independent of the order you pass them to the formatting function. This is especially useful when the logical flow of the sentence differs from the order of variables, or when reusing values multiple

times without repeating arguments. In Python, reordering is built into str.format() via explicit positional indices {0} , {1} , etc., and becomes even more natural with f-strings (Python 3.6+) through expressions. In 2026, while f-strings dominate for most formatting, understanding reordering in .format() remains valuable for dynamic templates, legacy code, internationalized strings, and c...

Category: Regular Expressions • From: Reordering values

Q8. What are the best practices for memoryview with TensorFlow in Python 2026: Zero-Copy NumPy → Tensor Interop + ML Examples in modern Python development?

memoryview with TensorFlow in Python 2026: Zero-Copy NumPy → Tensor Interop + ML Examples TensorFlow and NumPy have excellent interoperability in 2026 — you can often share memory between `np.ndarray` and `tf.Tensor` with zero or minimal copying. Adding memoryview lets you create efficient, zero-copy views/slices of large NumPy arrays before passing them to TensorFlow, which is especially valuable for memory-intensive tasks like image preprocessing, large batch handling, or data pipelines where duplicating gigabyte-scale arrays would crash or slow training. I've used this pattern in production CV models and time-series pipelines — slicing 4–8 GB image datasets for augmentation or feeding sub-regions direc...

Category: built in function • From: memoryview with TensorFlow in Python 2026: Zero-Copy NumPy → Tensor Interop + ML Examples

Q9. How does enumerate() in Python 2026: Index + Value Iteration + Modern Patterns & Best Practices work? Give a practical example.

enumerate() in Python 2026: Index + Value Iteration + Modern Patterns & Best Practices The built-in enumerate() function adds a counter (index) to an iterable and returns it as an iterator of tuples — the most elegant and Pythonic way to loop over items while knowing their position. In 2026 it remains one of the most frequently used built-ins for clean, readable iteration — especially in data processing, list comprehension, ML batch indexing, parallel processing, and UI rendering. With Python 3.12–3.14+ delivering faster iteration, better type hinting for enumerate (improved generics), and free-threading compatibility for concurrent loops, enumerate() is more powerful and type-safe than ever. This March 2...

Category: Built in Function • From: enumerate() in Python 2026: Index + Value Iteration + Modern Patterns & Best Practices

Q10. How does Aggregating multidimensional arrays work? Give a practical example.

Aggregating multidimensional arrays is a core operation in numerical computing — computing summary statistics (sum, mean, std, min/max, etc.) across one or more axes of NumPy, Dask, or xarray arrays. It reduces dimensions, extracts insights (e.g., global means, per-channel stats), and prepares data for

visualization or modeling. In 2026, aggregation is essential for large-scale analysis — climate averages over time/lat/lon, image channel means, time series rolling stats, ML feature reduction — where correct axis selection, chunk alignment (Dask), and lazy evaluation (xarray/Dask) determine speed, memory usage, and correctness. NumPy executes eagerly, Dask lazily with parallel reduction, xarray with labeled axe...

Category: Parallel Programming With Dask • From: Aggregating multidimensional arrays

Q11. Explain 'Creating timedeltas' in detail. Why is it important in 2026?

Creating timedeltas is a core skill for working with time differences in Python — the timedelta class from the datetime module lets you represent durations as days, seconds, and microseconds, then use them to shift dates/times, compute intervals, or measure elapsed time. In 2026, timedelta remains the standard for date/time arithmetic — essential for scheduling, age calculations, timeouts, performance benchmarking, data filtering by time windows, and any logic involving durations or offsets. It’s precise, handles edge cases like leap seconds (when used with aware datetimes), and integrates seamlessly with pandas/Polars for vectorized operations on large timestamp data. Here’s a complete, practical guid...

Category: Dates and Time • From: Creating timedeltas

Q12. Explain 'enumerate() and unpack' in detail. Why is it important in 2026?

enumerate() and unpack together form one of Python’s most powerful and readable iteration patterns — combining position tracking with flexible unpacking of tuple elements. The enumerate() function pairs each item from an iterable with its index, returning tuples like (0, item) . The unpacking operator * lets you split those tuples into separate variables — capturing the index in one variable and the rest (the item, or even sub-elements) in another. This is especially useful when the iterable yields tuples or when you want to handle the index separately from the data. In 2026, this combination is used constantly — for numbered lists, processing indexed data, unpacking coordinates, splitting rows from CSV...

Category: Data Science Tool Box • From: enumerate() and unpack

Q13. What are the best practices for aiter() in Python 2026: Asynchronous Iterator Protocol & Modern Async Usage in modern Python development?

aiter() in Python 2026: Asynchronous Iterator Protocol & Modern Async Usage The built-in aiter() function (introduced in Python 3.10) returns an asynchronous iterator for an async-iterable object — the async equivalent of iter() . In 2026, with async/await deeply integrated into web frameworks (FastAPI, Starlette), data streaming, concurrent I/O, and event-driven systems, aiter() is essential for clean, type-safe async iteration over streams, queues, generators, and third-party async iterables. Python 3.12–3.14+ made async iteration even more robust (better type hints, free-threading compatibility, improved asyncio performance), and aiter() is now commonly used in production async pipelines, WebSocke...

Category: Built in Function • From: aiter() in Python 2026: Asynchronous Iterator Protocol & Modern Async Usage

Q14. Explain 'Functions as objects' in detail. Why is it important in 2026?

Functions as objects is one of Python’s most powerful features — functions are first-class citizens, meaning they can be assigned to variables, passed as arguments, returned from other functions, stored in data structures, and treated like any other object (int, list, dict, etc.). This enables higher-order functions, callbacks, decorators, functional programming patterns, dynamic behavior, and code that is more flexible, reusable, and testable. In 2026, treating functions as objects remains essential — it powers decorators (logging, timing, caching), callbacks in GUIs/async I/O/ML pipelines, strategy patterns, dependency injection, and composable data transformations in pandas/Polars workflows. Understanding t...

Category: Writing Functions • From: Functions as objects

Q15. Explain 'Python AI in 2026 – Complete Guide to Building Intelligent Applications' in detail. Why is it important in 2026?

Python AI in 2026 has reached an incredible level of maturity. What started as simple machine learning scripts has evolved into powerful autonomous agents, multimodal systems, RAG applications, and production-grade AI solutions. Python remains the undisputed leader in AI development due to its rich ecosystem, ease of use, and massive community support. State of Python AI in March 2026 LLMs & Foundational Models : Grok-3, Claude 4, GPT-4.5, Llama 4, DeepSeek R1 Agentic AI : The biggest trend — AI agents that can plan, reason, use tools, and execute multi-step tasks autonomously Multimodal AI : Models that understand text + image + audio + video RAG Systems : Retrieval-Augmented Generation has beco...

Category: Agentic AI • From: Python AI in 2026 – Complete Guide to Building Intelligent Applications

Q16. Explain 'Dropping duplicate pairs' in detail. Why is it important in 2026?

Dropping duplicate pairs (or combinations of columns) is a critical cleaning step when dealing with real-world data — think duplicate customer records, repeated transactions, or merged datasets with overlapping entries. In Pandas, the .drop_duplicates() method lets you remove rows that have identical values in one or more specified columns (a "pair" or "group" of columns), while keeping control over which duplicate to retain. In 2026, this operation is essential for data quality in analytics, machine learning prep, and reporting. Here’s a practical guide with real examples. 1. Basic Setup & Sample Data import pandas as pd data = { 'Name': ['John', 'Mary', 'Peter', 'Anna', 'John', 'Mike'], ...

Category: Data Manipulation • From: Dropping duplicate pairs

Q17. Explain 'WebGL + AudioContext Integration Spoofing Techniques 2026 – Advanced Python Web Scrapping Evasion' in detail. Why is it important in 2026?

In 2026, the most sophisticated anti-bot systems no longer check Canvas, WebGL, or AudioContext in isolation. They analyze the **integration** and consistency between these fingerprints. WebGL + AudioContext integration spoofing has become one of the most powerful advanced evasion techniques for Python web scrapping. This guide shows how modern anti-bot platforms detect inconsistencies between WebGL and AudioContext, and provides battle-tested techniques to spoof their integration using Nodriver in 2026. Why WebGL + AudioContext Integration Matters Advanced anti-bot systems now look for natural correlations between different fingerprint vectors. If your Canvas/WebGL spoofing looks perfect but your Audio...

Category: Web Scrapping • From: WebGL + AudioContext Integration Spoofing Techniques 2026 – Advanced Python Web Scrapping Evasion

Q18. Explain 'Composing functions' in detail. Why is it important in 2026?

Composing functions is a core technique in functional programming that lets you build complex operations by chaining simple, reusable functions — each taking an input, transforming it, and passing the result to the next. In Python, function composition creates elegant, readable data processing pipelines, avoids intermediate variables/lists, and promotes modularity/testability. In 2026, composition remains essential — powering clean ETL, feature engineering, data validation, ML preprocessing, and Polars/pandas method chaining, where small pure functions combine into powerful transformations without side effects or memory bloat. Here’s a complete, practical guide to composing functions in Python: manual compos...

Category: Parallel Programming With Dask • From: Composing functions

Q19. Explain 'Learn Python in 2026: Complete Beginner Tutorial + Roadmap from Zero to Pro' in detail. Why is it important in 2026?

Learn Python in 2026: Complete Beginner Tutorial + Step-by-Step Roadmap from Zero to Pro Python remains the #1 programming language in 2026 — easy to read, incredibly powerful, and used everywhere: AI tools, data analysis (with Polars exploding), web apps (FastAPI), automation, scripting, and more. If you're starting from zero today (March 17, 2026), this guide gives you a realistic, updated roadmap to go from "Hello World" to confident coding — including free resources, practice projects, and 2026-specific tips (uv for fast installs, type hints, modern tools). I've taught Python to beginners and watched many land their first jobs or side projects in 3–9 months. The key? Consistent daily practice + building...

Category: Introduction • From: Learn Python in 2026: Complete Beginner Tutorial + Roadmap from Zero to Pro

Q20. What are the best practices for ord() in Python 2026: Unicode Code Point from Character + Modern Use Cases & Best Practices in modern Python development?

ord() in Python 2026: Unicode Code Point from Character + Modern Use Cases & Best Practices The built-in ord() function returns the Unicode code point (integer) of a single character string. In 2026 it remains the standard way to convert characters to their numeric code points — essential for text processing, encoding/decoding, cryptography (char → int mapping), tokenization in ML/NLP, Unicode debugging, and low-level string manipulation. With Python 3.12–3.14+ offering faster Unicode handling, full support for Unicode 15.1+, better free-threading safety for string operations, and growing use in multilingual AI and emoji processing, ord() is more relevant than ever. This March 24, 2026 update explains how...

Category: Built in Function • From: ord() in Python 2026: Unicode Code Point from Character + Modern Use Cases & Best Practices

Q21. Explain 'Timezone Hopping with Pendulum: Seamlessly Manage Time across Different Timezones' in detail. Why is it important in 2026?

Timezone Hopping with Pendulum: Seamlessly Manage Time across Different Timezones turns one of programming's trickiest challenges — accurate, DST-safe, human-friendly timezone conversions — into something fluent, reliable, and enjoyable. Pendulum extends Python's datetime with a chainable, intuitive API that handles parsing, timezone switching, offset calculation, DST transitions, and relative time formatting effortlessly. In 2026, Pendulum remains a favorite for apps needing clean global time logic (APIs, logging, scheduling, user-facing dates, time-series) — especially when you want to avoid manual offset math, DST bugs, or verbose zoneinfo / pytz code. It integrates beautifully with Polars/pandas/Dask f...

Category: Datatypes • From: Timezone Hopping with Pendulum: Seamlessly Manage Time across Different Timezones

Q22. What are the best practices for Combinations with loop in modern Python development?

Combinations with loop are a classic way to generate all possible selections of items from a sequence without regard to order — useful for picking subsets, creating pairs/groups, testing combinations, or solving combinatorial problems. In Python, nested loops with careful indexing let you build combinations manually, but for larger sizes or more combinations, this becomes slow, memory-heavy, and error-prone. In 2026, while loops teach the concept, itertools.combinations() is the go-to for efficiency and readability — it's lazy, memory-safe, and implemented in C for speed. Here's a complete, practical guide to generating combinations with loops and beyond: manual nested loops, common patterns, performance lim...

Category: Efficient Code • From: Combinations with loop

Q23. Explain 'HELP! Libraries to Make Python Development Easier' in detail. Why is it important in 2026?

Libraries to Make Python Development Easier — Python’s true power comes from its ecosystem. With thousands of libraries, you can avoid reinventing the wheel and focus on solving real problems. In 2026, the landscape has matured: Polars has overtaken pandas for speed & memory efficiency in data work, FastAPI dominates modern web APIs, Pydantic is the standard for data validation & settings, Typer / Click power clean CLIs, and tools like Rich , Loguru , HTTPX , Structlog , and Poetry make everyday development faster, prettier, and more reliable. This guide covers the must-have libraries across categories — with real-world use cases, why they matter in 2026, and how they fit into modern workflows ...

Category: Datatypes • From: HELP! Libraries to Make Python Development Easier

Q24. How does Adding time to the mix work? Give a practical example.

Adding time to the mix with strftime() unlocks the full power of formatting datetime objects — combining date and time components into human-readable or machine-friendly strings using flexible format codes. While date-only formats like %Y-%m-%d are great for calendars and reports, including time (%H:%M:%S , %I:%M %p) is essential for logs, timestamps, event scheduling, API payloads, and user-facing displays. In 2026, mastering time-inclusive strftime() remains critical — it ensures consistent, parsable output for databases, JSON, CSV exports, notifications, and analytics dashboards, often combined with time zones and vectorized pandas/Polars operations for large datasets. Here’s a complete, practi...

Category: Dates and Time • From: Adding time to the mix

Q25. How does Attaching nonlocal variables to nested functions work? Give a practical example.

Attaching nonlocal variables to nested functions via default arguments is a subtle but useful Python technique — it captures a snapshot of an outer scope variable’s value at the moment the inner function is defined, binding that value as the default for an inner parameter. This avoids the nonlocal keyword while still providing access to outer values — but importantly, it captures a copy of the value (not a live reference), so modifications inside the inner function affect only the local parameter, not the original outer variable. In 2026, this pattern is mostly legacy or niche (e.g., creating many similar closures with different static values), and modern Python strongly prefers explicit nonlocal , factory ...

Category: Writing Functions • From: Attaching nonlocal variables to nested functions

Q26. Explain 'Building Dask Bags & Globbing' in detail. Why is it important in 2026?

Building Dask Bags & Globbing is a flexible, powerful way to handle large-scale non-tabular or semi-structured data (text files, JSON, log lines, binary blobs, custom records) in parallel — especially when data doesn’t fit neatly into DataFrames or Arrays. Dask Bags treat each file or line as an independent element, allowing map/filter/reduce operations across millions of items without loading everything into memory. Globbing with from_filenames() or read_text() makes it easy to process partitioned directories or cloud storage. In 2026, Dask Bags remain essential for ETL on raw logs, text corpora, JSONL datasets,

sensor streams, or pre-processed earthquake catalogs — complementing Dask DataFrames (tabular)...

Category: Parallel Programming With Dask • From: Building Dask Bags & Globbing

Q27. Explain 'The timer decorator' in detail. Why is it important in 2026?

The timer decorator is one of the most practical and widely used examples of decorators in Python — it wraps any function to automatically measure and log its execution time without modifying the original code. By using time.perf_counter() (high-resolution wall-clock time), it provides accurate, real-world performance insights — ideal for profiling data pipelines, ML model inference, web endpoints, batch jobs, or any performance-critical function. In 2026, timer decorators remain essential — they separate timing/observability from business logic, enable easy application to multiple functions via @timer , support exception-safe cleanup, and integrate seamlessly with logging, pandas/Polars workflows, and prod...

Category: Writing Functions • From: The timer decorator

Q28. Explain 'Querying DataFrame memory usage' in detail. Why is it important in 2026?

Querying DataFrame memory usage in Python is essential for optimizing performance, debugging memory issues, and ensuring scalability — especially with large datasets in pandas or Polars pipelines. Pandas provides memory_usage() for per-column and total usage (including deep object memory with deep=True), while Polars offers estimated_size() and memory_usage() for columnar, compressed estimates. In 2026, accurate memory profiling helps choose dtypes (int32/float32 vs int64/float64), detect leaks, compare pandas vs Polars efficiency, and prevent OOM errors in production. Combine with psutil for process-level RSS/USS and tracemalloc for allocation tracking to get the full picture. Here’s a complete,...

Category: Parallel Programming With Dask • From: Querying DataFrame memory usage

Q29. How does Humanizing Differences: Making Time Intervals More Readable with Pendulum work? Give a practical example.

Humanizing Differences: Making Time Intervals More Readable with Pendulum transforms raw time deltas — seconds, minutes, days — into natural, human-friendly strings like "3 hours ago", "in 2 weeks", or "yesterday at 2:35 PM". Pendulum’s diff_for_humans() (and in_words()) method handles past/future orientation, granularity control, locale support, and localization automatically — making timestamps, logs, notifications, and user-facing dates instantly understandable. In 2026, this feature is invaluable for dashboards, chat apps, activity feeds, monitoring tools, and time-series UIs — especially when paired with Polars/pandas/Dask for columnar age calculations, FastAPI for API responses, and Pydantic for vali...

Category: Datatypes • From: Humanizing Differences: Making Time Intervals More Readable with Pendulum

Q30. Explain 'max() in Python 2026: Finding Maximum Values + Modern Patterns & Best Practices' in detail. Why is it important in 2026?

max() in Python 2026: Finding Maximum Values + Modern Patterns & Best Practices The built-in max() function returns the largest item in an iterable or the largest of two or more arguments. In 2026 it remains one of the most frequently used built-ins for finding extremes — essential in data analysis, ML model selection, optimization, ranking algorithms, validation, and everyday decision logic. With Python 3.12–3.14+ improving performance for large iterables, better type hinting for comparable types, and free-threading compatibility for concurrent max operations, max() is faster and safer in modern code. This March 24, 2026 update covers how max() works today, real-world patterns, key argument usage, perfor...

Category: Built in Function • From: max() in Python 2026: Finding Maximum Values + Modern Patterns & Best Practices

Q31. Explain 'Cost Optimization Techniques for Multi-Agent Systems in 2026' in detail. Why is it important in 2026?

Running multi-agent AI systems can become extremely expensive very quickly in 2026. A single complex agent workflow can easily consume thousands of tokens per request. Without proper cost optimization strategies, production Agentic AI systems can quickly become financially unsustainable. This practical guide covers proven cost optimization techniques for multi-agent systems built with CrewAI, LangGraph, and other frameworks as of March 24, 2026. Why Cost Optimization is Critical in 2026 Modern agentic systems often involve: Multiple LLM calls per task Long context windows Tool usage and external API calls Persistent memory and vector search operations Without optimization, costs can spiral o...

Category: Agentic AI • From: Cost Optimization Techniques for Multi-Agent Systems in 2026

Q32. How does bytes() in Python 2026: Immutable Binary Sequences + Modern Use Cases & Best Practices work? Give a practical example.

bytes() in Python 2026: Immutable Binary Sequences + Modern Use Cases & Best Practices The built-in bytes() type creates an immutable sequence of bytes (values 0–255) — the read-only counterpart to bytearray . In 2026 it remains the standard for fixed binary data: keys, hashes, network payloads, file contents, image bytes, crypto material, and protocol messages where immutability guarantees safety and hashability. With Python 3.12–3.14+ offering faster bytes operations, better memoryview interop, and free-threading compatibility, bytes objects are more efficient than ever in concurrent I/O, streaming, and ML binary preprocessing. This March 23, 2026 update explains how bytes() works today, creation patte...

Category: Built in Function • From: bytes() in Python 2026: Immutable Binary Sequences + Modern Use Cases & Best Practices

Q33. Explain 'Calculating summary stats across columns' in detail. Why is it important in 2026?

Calculating summary statistics across columns is one of the fastest ways to get a high-level view of your data in pandas — mean, median, standard deviation, min/max, sum, and more, all computed column-wise in a single call. These operations help you spot trends, outliers, distributions, and scale differences between variables before diving deeper into modeling or visualization. In 2026, this remains a core EDA step — quick, readable, and essential for understanding wide datasets. Here’s a practical guide with real examples you can copy and adapt. 1. Basic Setup & Sample Data import pandas as pd import numpy as np df = pd.DataFrame({ 'A': [1, 2, 3, 4, 5], 'B': [6, 7, 8, 9, 10], 'C': [...

Category: Data Manipulation • From: Calculating summary stats across columns

Q34. How does Group by to pivot table work? Give a practical example.

Converting grouped data into a pivot table is a common and powerful workflow in Pandas — you first use groupby() to aggregate raw data, then reshape the result into a clean, cross-tabulated pivot table for reporting, dashboards, or further analysis. This combination gives you the flexibility of groupby with the readability of pivot tables (similar to Excel, but fully programmable). In 2026, this pattern remains essential for sales breakdowns, cohort reports, A/B test summaries, and multi-dimensional views. Here’s a practical guide with real examples you can copy and adapt. 1. Basic Setup & Sample Data import pandas as pd data = { 'Region': ['North', 'North', 'South', 'South', 'West', 'West'],...

Category: Data Manipulation • From: Group by to pivot table

Q35. Explain 'Negative look-behind' in detail. Why is it important in 2026?

Negative look-behind is a zero-width assertion in Python’s re module that matches a position in the string only if it is NOT immediately preceded by a specified pattern — without consuming or including the preceding text in the match. The syntax is (? , where pattern is the forbidden prefix that must not appear before the match. Negative look-behind is ideal for exclusionary matching from the left: match something only if it does not come right after an unwanted marker, such as numbers not preceded by currency symbols, words not after titles, or tags not after certain opening delimiters. In 2026, negative look-behind remains a powerful regex feature — essential in data extraction, validation, cleaning (e....

Category: Regular Expressions • From: Negative look-behind

Q36. What are the best practices for memoryview() in Python 2026: Zero-Copy Memory Views + Modern Use Cases & Best Practices in modern Python development?

memoryview() in Python 2026: Zero-Copy Memory Views + Modern Use Cases & Best Practices The built-in memoryview() function creates a memory view object — a safe, zero-copy view into the memory buffer of an object that supports the buffer protocol (bytes, bytearray, array.array, mmap, NumPy arrays, etc.). In 2026 it remains one of the most powerful tools for high-performance binary data handling — essential in large file processing, network packet parsing, image/video manipulation, ML preprocessing,

and interop with C extensions or low-level I/O without unnecessary copying. With Python 3.12–3.14+ offering faster buffer protocol operations, better memoryview interop with NumPy/JAX/PyTorch, and free-threadin...

Category: Built in Function • From: memoryview() in Python 2026: Zero-Copy Memory Views + Modern Use Cases & Best Practices

Q37. How does Sets for Unordered and Unique Data with Tuples in Python work? Give a practical example.

Sets for Unordered and Unique Data with Tuples in Python combine two of Python's most powerful features: sets (unordered, unique, hash-based collections) and tuples (immutable, ordered sequences). Because tuples are hashable (when their elements are hashable), they can serve as set elements or dictionary keys — enabling you to store complex, composite records while guaranteeing uniqueness and fast membership testing. In 2026, this pattern is widely used in data science (deduplication of multi-attribute events, unique coordinate sets), software engineering (unique user sessions, tag combinations), and algorithms (graph edges, unique configurations) — especially with Polars/Dask for large-scale unique value mana...

Category: Datatypes • From: Sets for Unordered and Unique Data with Tuples in Python

Q38. Explain 'Python No-GIL (Free-Threaded) vs Rust in 2026 - Performance, Concurrency & When to Choose Each' in detail. Why is it important in 2026?

Updated March 12, 2026 : Covers DuckDB 1.2+ (embedded analytics engine), Polars 1.x (lazy/streaming DataFrame), real-world benchmarks on 100M–1B row datasets (single-node M-series & AMD hardware), SQL vs expression API comparison, in-memory vs file-based performance, uv-based install, and current 2026 recommendations. All timings aggregated from community benchmarks & official blogs (March 2026). DuckDB vs Polars in 2026 – Which is Better for Fast Analytics? (Benchmarks + Guide) In 2026, two of the most exciting tools for fast, in-process analytics are DuckDB (embedded SQL OLAP database) and Polars (high-performance DataFrame library with lazy evaluation). Both are written in Rust/C++, both are blazing fast...

Category: Efficient Code • From: Python No-GIL (Free-Threaded) vs Rust in 2026 - Performance, Concurrency & When to Choose Each

Q39. How does Don't repeat yourself (DRY) work? Give a practical example.

"Don't repeat yourself" (DRY) is a software development principle that advocates for reducing repetition and redundancy in code. The idea is that if you have duplicate code or logic in your program, you increase the risk of errors and make it harder to maintain and update your code. The DRY principle suggests that you should create reusable code components that can be called multiple times from different parts of your program. This can be achieved through various techniques, such as modularization, inheritance, and

functional programming. By avoiding duplication, you can simplify your code, reduce its size, and make it more readable and maintainable. DRY is closely related to other software engineering prin...

Category: Writing Functions • From: Don't repeat yourself (DRY)

Q40. How does Passing an incorrect argument work? Give a practical example.

Passing an incorrect argument to a function is one of the most common sources of runtime errors in Python. When the type, value, or number of arguments doesn't match what the function expects, you'll typically see a TypeError , ValueError , or unexpected behavior — and in production code, these can crash your program or produce silent bugs. In 2026, robust functions use type hints, input validation, and proper exception handling to detect and respond to bad arguments gracefully. Here's a complete, practical guide to understanding what happens when you pass wrong arguments, how to catch and handle these issues, and modern best practices to make your code resilient and user-friendly. Start with the classic...

Category: Data Science Tool Box • From: Passing an incorrect argument

Q41. Explain 'Timing array computations' in detail. Why is it important in 2026?

Timing array computations is essential for benchmarking NumPy and Dask array performance, identifying bottlenecks, optimizing chunk sizes, comparing implementations, and ensuring scalability in large-scale numerical workflows. In 2026, accurate timing helps validate whether Dask parallelism provides speedup over NumPy, guides chunking strategy (too small = overhead, too large = memory issues), and reveals I/O vs compute trade-offs in out-of-core processing. Use time.perf_counter() for wall-clock time, time.process_time() for CPU time, timeit for micro-benchmarks, and Dask diagnostics for detailed task-level profiling. Combine with psutil for memory tracking during timed operations. Here's a complete,...

Category: Parallel Programming With Dask • From: Timing array computations

Q42. What are the best practices for Building with Builtins in Python 2026: Write Faster & Cleaner Code in modern Python development?

Building with Builtins in Python 2026: Write Faster & Cleaner Code Python's built-in functions and types are highly optimized in C and form the foundation of efficient code. In 2026, mastering builtins is one of the quickest ways to write faster, more readable, and more Pythonic code without adding external dependencies. This March 15, 2026 update shows how to leverage Python builtins for common tasks, performance-critical operations, and modern patterns in 2026. TL;DR — Key Takeaways 2026 Builtins are faster than custom loops or external libraries for most operations Use len() , sum() , max() , min() , any() , all() instead of manual loops Prefer dict.get() , collections.defaultdict , an...

Category: Efficient Code • From: Building with Builtins in Python 2026: Write Faster & Cleaner Code

Q43. What are the best practices for Filling missing values in pivot tables in modern Python development?

Filling missing values in pivot tables is a critical step when reshaping data — especially in cross-tab reports where some combinations (e.g., region × month × salesperson) have no data, resulting in NaN. Pandas' pivot_table() makes this easy with the fill_value parameter, which replaces missing entries with a specified value (usually 0) to make tables cleaner and calculations more reliable. In 2026, this is essential for dashboards, financial reporting, cohort analysis, and any multi-dimensional view. Here's a practical guide with real examples you can copy and adapt. 1. Basic Setup & Sample Data import pandas as pd data = { 'Region': ['North', 'North', 'South', 'South', 'West', 'West'], ...

Category: Data Manipulation • From: Filling missing values in pivot tables

Q44. What are the best practices for Why Should We Time Our Code in Python 2026 with Efficient Code in modern Python development?

Why Should We Time Our Code in Python 2026 with Efficient Code Timing your code is one of the most important habits for writing truly efficient Python programs. In 2026, with free-threading, faster interpreters, and increasingly complex applications, knowing exactly how long your code takes to run is no longer optional — it's essential for performance optimization and making informed decisions. This March 15, 2026 guide explains why timing code matters and shows modern best practices for measuring performance in Python. TL;DR — Key Takeaways 2026 Timing reveals real performance bottlenecks that intuition often misses Premature optimization is the root of all evil — timing prevents it Modern tools ...

Category: Efficient Code • From: Why Should We Time Our Code in Python 2026 with Efficient Code

Q45. What are the best practices for Additional datetime methods in Pandas in modern Python development?

Summarizing datetime data in pandas is a core skill for time-series analysis — it lets you aggregate, group, and resample data by periods (day, week, month, quarter, year) to uncover trends, seasonality, daily patterns, or performance metrics. Pandas provides two powerful tools: groupby() with pd.Grouper(freq=...) for flexible grouping, and resample() for high-performance time-based resampling on datetime-indexed DataFrames. In 2026, these methods remain essential — especially for large datasets in finance, IoT, user analytics, weather, sales forecasting, or any time-indexed data — and Polars offers even faster, more memory-efficient alternatives for massive scale. Here's a complete, practical guide to...

Category: Dates and Time • From: Additional datetime methods in Pandas

Q46. Explain 'Using iterators to load large files into memory' in detail. Why is it important in 2026?

Using iterators to load large files into memory is one of the most important techniques in Python for handling massive datasets — logs, CSVs, JSONL, genomic data, or any file too big to fit in RAM. Instead of loading everything at once (which crashes on gigabytes+ files), iterators read the file incrementally — line by line, chunk by chunk, or record by record — using constant memory. In 2026, this pattern is essential for production data pipelines, streaming processing, and working with terabyte-scale data on modest machines. Here's a complete, practical guide to using iterators for large files: line-by-line iteration, chunked reading, real-world patterns with CSV/JSONL, generators, and modern best practice...

Category: Data Science Tool Box • From: Using iterators to load large files into memory

Q47. What are the best practices for Working with Dictionaries More Pythonically: Efficient Data Manipulation in modern Python development?

Working with Dictionaries More Pythonically: Efficient Data Manipulation is one of the hallmarks of idiomatic Python — leveraging built-in features like comprehensions, unpacking, defaultdict, zip, and modern syntax to write concise, readable, and performant dictionary code. In 2026, these techniques are even more powerful with typed dicts (TypedDict), Pydantic models, Polars/Dask data pipelines, and runtime config systems that demand clean, dynamic key-value handling. This guide covers the most Pythonic ways to create, transform, merge, filter, and manipulate dictionaries — with real-world patterns, performance notes, and integration with modern data tools. Here's a complete, practical guide to Pythonic dic...

Category: Datatypes • From: Working with Dictionaries More Pythonically: Efficient Data Manipulation

Q48. What are the best practices for Visualizing a task graph in modern Python development?

Visualizing a task graph in Dask is one of the most powerful debugging and optimization tools — it reveals the exact structure of your lazy computation (nodes for tasks, edges for dependencies), helping you spot bottlenecks, redundant work, unnecessary serialization, poor chunking, or opportunities for parallelism. In 2026, task graph visualization remains essential for anyone using Dask at scale — whether on single machines (threads/processes) or distributed clusters — it turns abstract Delayed , dask.array , or dask.dataframe operations into clear diagrams you can inspect before (or after) .compute() . Use it to validate pipelines, debug failures, optimize chunk sizes, and understand why a computation i...

Category: Parallel Programming With Dask • From: Visualizing a task graph

Q49. What are the best practices for Detecting any missing values in modern Python development?

Detecting any missing values with .isna().any() is the fastest 1-line sanity check you'll ever write. It answers the critical question: "Does this column (or the whole DataFrame) contain even a single NaN/None/null?" — without printing huge boolean masks or computing full counts. In pandas, df.isna().any() returns a boolean Series: True for every column that has at least one missing value, False otherwise. Chain another .any() to

check the entire dataset in one boolean. In 2026, this is still your go-to first line after pd.read_csv() or pl.read_csv() . 1. Basic & Practical Usage (Pandas) import pandas as pd # Realistic messy dataset example data = { 'name': ['Alice', 'Bob', 'Charlie...

Category: Data Manipulation • From: Detecting any missing values

Q50. Explain 'Iterating and Sorting' in detail. Why is it important in 2026?

Iterating and Sorting Lists are two of the most essential daily operations when working with data in Python. Iteration lets you process each element — apply transformations, filter, aggregate, or extract insights. Sorting organizes data for display, ranking, searching, or analysis. In 2026, these operations power everything from simple scripts to large-scale data pipelines in pandas, Polars, and Dask. Python offers elegant, efficient, and expressive ways to iterate (for loops, comprehensions, enumerate, zip) and sort (sorted(), .sort(), key functions, custom comparators) — often vectorized or parallelized in modern libraries. Here’s a complete, practical guide to iterating and sorting lists in Python: core i...

Category: Datatypes • From: Iterating and Sorting

Q51. How does DateTime Components work? Give a practical example.

Understanding datetime components is essential in Python — whether you're working with logs, timestamps, schedules, financial data, or real-time AI applications. The datetime object gives you full access to every part of a date and time, and in 2026, knowing how to extract and manipulate these components efficiently is a core skill. Core Components of a datetime Object A datetime object contains these main attributes: year — 4-digit year (e.g. 2026) month — 1–12 day — 1–31 hour — 0–23 minute — 0–59 second — 0–59 microsecond — 0–999999 tzinfo — timezone info (None for naive, ZoneInfo for aware) fold — handles DST ambiguity (rarely used) from datetime import datetime ...

Category: Data Manipulation • From: DateTime Components

Q52. Explain 'run_n_times()' in detail. Why is it important in 2026?

The run_n_times() decorator is a classic parameterized decorator example in Python — it takes an integer n and returns a decorator that wraps the target function to execute it n times on each call. This pattern is useful for repeated execution (e.g., stress testing, Monte Carlo simulations, retry logic with fixed count, or benchmarking average performance), and it demonstrates the decorator factory pattern: outer function accepts parameters, returns inner decorator, which returns wrapper. In 2026, @run_n_times(n) remains a go-to teaching example and real-world utility — often extended for logging each run, collecting results, handling exceptions, or integrating with pandas/Polars for repeated data transf...

Category: Writing Functions • From: run_n_times()

Q53. Explain 'Producing a visualization of data_dask for Analyzing Earthquake Data' in detail. Why is it important in 2026?

Producing a visualization of data_dask for Analyzing Earthquake Data turns massive, chunked seismic catalogs into interpretable plots — revealing spatial patterns, magnitude distributions, depth trends, and temporal clusters without loading the entire dataset into memory. Dask arrays enable lazy computation, so visualizations require smart sampling, aggregation, or reduction before plotting. In 2026, this workflow is standard for geophysics and disaster analysis — using .compute() on subsets, matplotlib/seaborn for static plots, hvplot/holoviews for interactive exploration, and cartopy for geospatial maps — all while leveraging Dask's parallel power and the USGS catalog's rich metadata (time, lat, lon, depth...

Category: Parallel Programming With Dask • From: Producing a visualization of data_dask for Analyzing Earthquake Data

Q54. What are the best practices for Lists and dictionaries of functions in modern Python development?

Lists and dictionaries of functions leverage Python's first-class function objects — allowing you to store functions in lists, dictionaries, or other structures for dynamic dispatch, strategy patterns, plugin systems, callbacks, batch processing, or configurable workflows. This turns functions into data: select which one to call based on keys, indices, conditions, or user input, making code highly flexible, extensible, and DRY. In 2026, this pattern is everywhere — dynamic operation selection in data pipelines, ML model ensembles, API handlers, GUI button actions, pandas/Polars custom transformations, and plugin architectures. It eliminates long if-elif chains and enables runtime behavior changes without rewri...

Category: Writing Functions • From: Lists and dictionaries of functions

Q55. How does Mastering Crawling & Pagination in Scrapy 2026 – Complete Python Web Scrapping Guide work? Give a practical example.

Crawling is the heart of any serious web scrapping project. In Scrapy (still the #1 framework for structured crawling in 2026), crawling means systematically following links across pages, handling pagination, respecting depth limits, and extracting data at scale — all while avoiding blocks and staying ethical. This updated 2026 guide explains how to build robust crawlers with Scrapy 2.14+, including modern async patterns, pagination strategies, CrawlSpider rules, depth control, and best practices to stay under the radar. What Does "Crawling" Mean in Web Scrapping? Crawling = discovering and visiting new pages by following hyperlinks. In Scrapy, this happens through: start_urls + parse() → manual f...

Category: Web Scrapping • From: Mastering Crawling & Pagination in Scrapy 2026 – Complete Python Web Scrapping Guide

Q56. What are the best practices for Using HDF5 files for analyzing earthquake data in modern Python development?

Using HDF5 files for analyzing earthquake data is a smart, scalable approach when dealing with large seismic catalogs — HDF5 offers hierarchical organization, compression, chunking, partial I/O, and rich metadata, making it ideal for storing time-indexed, multi-variable earthquake data (time, lat, lon, depth, magnitude, etc.) efficiently. In 2026, HDF5 remains the standard for big scientific datasets (USGS, IRIS, global catalogs), enabling fast queries, parallel access via h5py/Dask, labeled analysis with xarray, and integration with pandas/Polars for exploration or ML. This workflow avoids CSV bloat, supports compression (reduce file size 50–80%), and allows lazy loading of only needed slices — perfect for te...

Category: Parallel Programming With Dask • From: Using HDF5 files for analyzing earthquake data

Q57. What are the best practices for Adding and Extending Python Dictionaries: Flexible Data Manipulation in modern Python development?

Adding and Extending Python Dictionaries: Flexible Data Manipulation is one of the most frequent and powerful operations in Python — whether you're merging configuration layers, updating records from multiple sources, enriching earthquake metadata, building dynamic feature sets, or handling API responses. Dictionaries are mutable, so you can add, update, or combine them at runtime with several clean, expressive, and performant techniques. In 2026, these patterns are even more important with typed dicts (TypedDict), Pydantic models, Polars/Dask data pipelines, and runtime config systems that demand safe, dynamic key-value manipulation. Here’s a complete, practical guide to adding and extending dictionaries in...

Category: Datatypes • From: Adding and Extending Python Dictionaries: Flexible Data Manipulation

Q58. What are the best practices for Slice twice in modern Python development?

Double slicing — selecting both rows and columns at the same time — is one of the most common and powerful operations in pandas. It lets you filter rows based on conditions or positions, then immediately narrow to the exact columns you need, all in one clean line. In 2026, this pattern is essential for exploratory data analysis, feature selection, reporting, and preparing data for modeling. Here’s a practical guide with real examples using both .loc[] (label/condition-based) and .iloc[] (position-based). 1. Basic Setup & Sample Data import pandas as pd data = { 'Name': ['John', 'Mary', 'Peter', 'Anna', 'Mike', 'Sarah'], 'Age': [25, 32, 18, 47, 23, 29], 'Salary': [50000, 80000, 350...

Category: Data Manipulation • From: Slice twice

Q59. What are the best practices for Explicit indexes in modern Python development?

Explicit indexes are the foundation of element access in Python sequences — lists, tuples, strings, and more. An explicit index is simply a specific integer (starting from 0) that points to a particular position in the sequence, allowing you to read, modify, or slice data with precision. Understanding explicit indexing is essential for clean, efficient code — whether you're processing data, building algorithms, or working with

pandas DataFrames. Here's a practical guide with real examples. 1. Reading Elements with Explicit Indexes Indexes start at 0 (zero-based) — the first element is at index 0. my_list = [10, 20, 30, 40, 50] print(my_list[0]) # 10 (first element) print(my_list[2]) # 30 (thi...

Category: Data Manipulation • From: Explicit indexes

Q60. How does Introduction to the Scrapy Selector in Python 2026 work? Give a practical example.

Introduction to the Scrapy Selector in Python 2026 The Scrapy Selector is one of the most powerful and flexible tools for web scraping in Python. Built on top of parsel, it combines the best of CSS selectors and XPath, making it extremely efficient for extracting structured data from HTML and XML documents. In 2026, the Scrapy Selector remains a core component of the Scrapy framework and is widely used even in standalone scripts due to its speed, readability, and advanced features. This March 24, 2026 guide introduces the Scrapy Selector with modern best practices. TL;DR — Key Takeaways 2026 Selector from Scrapy (based on parsel) supports both CSS and XPath Use .css() for simple and readable sele...

Category: Web Scrapping • From: Introduction to the Scrapy Selector in Python 2026

Q61. How does Exploring the Collections Module in Python: Enhance Data Structures and Operations work? Give a practical example.

Exploring the collections Module in Python: Enhance Data Structures and Operations unlocks some of Python’s most powerful and Pythonic tools for advanced data handling. The collections module extends built-in types with specialized, high-performance alternatives — Counter for frequency counting, defaultdict for auto-initialization, deque for efficient queues/stacks, namedtuple for lightweight records, ChainMap for layered lookups, OrderedDict for insertion-order preservation, and more. In 2026, these remain essential in data science (frequency analysis, grouping), software engineering (caching, config layering, plugin systems), and performance-critical code — often outperforming plain dicts/lists...

Category: Datatypes • From: Exploring the Collections Module in Python: Enhance Data Structures and Operations

Q62. How does Nodriver Canvas Fingerprint Spoofing 2026 – Advanced Anti-Detection Techniques work? Give a practical example.

Canvas fingerprint spoofing is one of the most important advanced evasion techniques in 2026 for anyone doing serious web scrapping with Python. Anti-bot systems heavily rely on canvas fingerprinting to detect automation. Nodriver , being a lightweight and stealth-focused library, gives developers powerful control over canvas spoofing — often better than standard Playwright setups. This detailed guide explains how canvas fingerprinting works, why it matters in 2026, and how to implement effective canvas spoofing techniques using Nodriver. What is Canvas Fingerprinting? Canvas fingerprinting is a tracking technique where websites force the browser to draw text or images on a hidden HTML5 canvas element a...

Category: Web Scrapping • From: Nodriver Canvas Fingerprint Spoofing 2026 – Advanced Anti-Detection Techniques

Q63. Explain 'Conditionals in generator expressions' in detail. Why is it important in 2026?

Conditionals in generator expressions give you the same filtering and transformation power as list comprehensions — but with the memory efficiency and lazy evaluation of generators. You can add if clauses to include only items that meet a condition, or use conditional expressions (if/else) to yield different values based on a test — all while producing values one at a time on demand. In 2026, this pattern is essential for processing large or infinite streams, filtering data lazily, and building memory-efficient pipelines in data science, file handling, APIs, and streaming applications. Here’s a complete, practical guide to conditionals in generator expressions: filtering with if , value transformation w...

Category: Data Science Tool Box • From: Conditionals in generator expressions

Q64. How does Tuples work? Give a practical example.

Tuples are Python’s lightweight, immutable sequences — ordered collections of elements that cannot be changed after creation. They are faster, more memory-efficient, and hashable (if elements are hashable), making them perfect for fixed records, function returns, dictionary keys, set elements, and data that should never be accidentally modified. In 2026, tuples remain a cornerstone in data science (multi-index keys, coordinate pairs, pandas group keys), software engineering (named records via namedtuple, return values), and performance-critical code — offering O(1) access, hashability, and zero-copy slicing when possible. Here’s a complete, practical guide to using tuples in Python: creation patterns, unpack...

Category: Datatypes • From: Tuples

Q65. How does hash() in Python 2026: Object Hashing, Hashability & Modern Use Cases work? Give a practical example.

hash() in Python 2026: Object Hashing, Hashability & Modern Use Cases The built-in hash() function returns the hash value of an object — an integer used for fast lookup in dictionaries and sets. In 2026 it remains a fundamental part of Python’s hash table implementation and is critical for understanding hashability, custom hashing (__hash__), caching, deduplication, and performance in data structures. With Python 3.12–3.14+ introducing deterministic hashing in more contexts, better free-threading safety for hash tables, and improved type hinting for hashable types, hash() is more predictable and performant. This March 23, 2026 update explains hash() behavior, hashability rules, real-world patterns (cachin...

Category: Built in Function • From: hash() in Python 2026: Object Hashing, Hashability & Modern Use Cases

Q66. What are the best practices for Stripping characters in modern Python development?

Stripping characters is a fundamental string cleaning operation in Python — the strip() , lstrip() , and rstrip() methods remove leading and/or trailing characters from a string, defaulting to whitespace (spaces, tabs, newlines) when no argument is provided. You can also specify custom characters (or a set of characters) to strip, making it powerful for normalizing text, removing delimiters, cleaning CSV fields, log prefixes/suffixes, or preparing data for parsing. In 2026, stripping remains essential — especially in data preprocessing, text normalization, user input sanitization, and pandas/Polars column cleaning where vectorized .str.strip() scales to millions of rows efficiently. Here's a complete, ...

Category: Regular Expressions • From: Stripping characters

Q67. What are the best practices for OR operand in re module in modern Python development?

The OR operand in Python's re module — represented by the vertical bar | — allows a regular expression to match one pattern OR another (or more), giving you the ability to create flexible alternatives in a single pattern. It's the regex equivalent of logical OR and is one of the most powerful and frequently used features for handling variations — different keywords, formats, delimiters, or optional parts in text. In 2026, the OR operand remains essential — used constantly in data validation (multiple date/phone formats), text extraction (synonyms or variants), log parsing (different log levels), and vectorized pandas/Polars string operations where pattern alternatives scale efficiently across large dataset...

Category: Regular Expressions • From: OR operand in re module

Q68. What are the best practices for Look-behind in modern Python development?

Look-behind is a zero-width assertion in Python's re module that matches a position in the string only if it is immediately preceded by a specified pattern — without consuming or including the preceding text in the match. Positive look-behind (?<=pattern) succeeds only if the position is preceded by the pattern; negative look-behind (? succeeds only if it is not preceded by the pattern. Look-behind is ideal for context-dependent matching from the left: match something only if it comes after (or does not come after) a specific prefix, such as numbers after currency symbols, words after titles, or tags after opening delimiters. In 2026, look-behind remains a powerful regex feature — essential in data extra...

Category: Regular Expressions • From: Look-behind

Q69. How does Reading text files work? Give a practical example.

Reading text files into Dask is the gateway to scalable, parallel processing of large-scale unstructured or semi-structured text data — logs, documents, JSONL, CSV-like files, sensor streams, or any line-based format too big for memory. Dask's dask.bag.read_text() reads files lazily via glob patterns or lists, creating

a Bag where each line (or file) is an element that can be mapped, filtered, reduced, or converted to DataFrames in parallel across cores or clusters. In 2026, this remains essential for log analysis, text mining, NLP preprocessing, multi-file JSONL catalogs (e.g., earthquake metadata), and ETL pipelines — complementing Dask DataFrames (tabular) and Arrays (numerical) while scaling effortlessly...

Category: Parallel Programming With Dask • From: Reading text files

Q70. How does Chunking Arrays in Dask work? Give a practical example.

Chunking Arrays in Dask is the foundation of scalable, out-of-core array computing in Python — it splits massive NumPy-like arrays into smaller, manageable chunks that fit in memory, enabling parallel processing across cores or distributed clusters without loading the full array at once. In 2026, Dask array chunking powers high-performance numerical workflows — simulations, image processing, geospatial analysis, ML feature engineering, and large-scale scientific data — where arrays reach terabytes. Proper chunking balances parallelism (more chunks = more tasks), memory usage (chunks fit in RAM), and I/O efficiency (avoid tiny or huge chunks), while da.from_array , da.zeros , da.random , and rechunking give ...

Category: Parallel Programming With Dask • From: Chunking Arrays in Dask

Q71. Explain 'str() in Python 2026: String Conversion + Modern Formatting & Best Practices' in detail. Why is it important in 2026?

In Python, the str() function is a built-in function used to convert an object into a string. This function can be used to convert integers, floats, lists, tuples, dictionaries, and other data types to string data type. Syntax: The syntax of the str() function is as follows: str (object , encoding = 'utf-8' , errors = 'strict') Here, object is the object that needs to be converted to a string. The optional encoding and errors parameters specify the encoding to be used for the conversion and the error handling mechanism to be used if the conversion fails. Examples: Let's take a look at some examples of using the str() function: Convert an integer to a string: ...

Category: Built in Function • From: str() in Python 2026: String Conversion + Modern Formatting & Best Practices

Q72. Explain 'Using Dask DataFrames' in detail. Why is it important in 2026?

Using Dask DataFrames is one of the most powerful ways to scale pandas-like data analysis to datasets that are too large for memory or single-core processing. Dask DataFrames mimic the pandas API but execute lazily and in parallel — using chunked, distributed computation across multiple cores, machines, or clusters. In 2026, Dask DataFrames remain essential for big data ETL, exploratory analysis, feature engineering, and machine learning preprocessing — handling CSV/Parquet/HDF5 files in the gigabyte-to-terabyte range, with seamless integration with pandas (for small results), Polars (for single-machine speed), xarray (for labeled data), and distributed schedulers for true cluster scale. Here’s a complete, p...

Category: Parallel Programming With Dask • From: Using Dask DataFrames

Q73. Explain 'A Classy Spider in Python 2026: Building Web Crawlers with Elegance & Best Practices' in detail. Why is it important in 2026?

A Classy Spider in Python 2026: Building Web Crawlers with Elegance & Best Practices Building a web crawler (often called a "spider") is a classic Python project that teaches asynchronous I/O, data extraction, rate limiting, and respectful crawling. In 2026, with improved async support, better libraries (httpx, BeautifulSoup4, Playwright, Scrapy), and stricter ethical guidelines, writing a "classy spider" means creating clean, efficient, respectful, and maintainable crawlers. This March 24, 2026 update walks through building a modern, classy spider in Python using best practices: asynchronous requests, proper headers, rate limiting, data validation, error handling, and ethical considerations. TL;DR — Key ...

Category: Web Scrapping • From: A Classy Spider in Python 2026: Building Web Crawlers with Elegance & Best Practices

Q74. How does CSS Locators in Python 2026: Powerful Web Scraping Techniques work? Give a practical example.

CSS Locators in Python 2026: Powerful Web Scraping Techniques CSS locators (also called CSS selectors) are one of the fastest and most readable ways to locate elements during web scraping. In 2026, with modern async scraping tools and dynamic websites, CSS locators remain the go-to choice for most Python developers due to their simplicity, speed, and maintainability compared to XPath. This March 24, 2026 guide covers everything you need to know about using CSS locators effectively in Python web scraping with BeautifulSoup, parsel, httpx, and Playwright. TL;DR — Key Takeaways 2026 CSS locators are faster and cleaner than XPath in most cases Use .select() or .select_one() in BeautifulSoup Master...

Category: Web Scrapping • From: CSS Locators in Python 2026: Powerful Web Scraping Techniques

Q75. How does id() in Python 2026: Object Identity & Memory Address + Modern Introspection Use Cases work? Give a practical example.

id() in Python 2026: Object Identity & Memory Address + Modern Introspection Use Cases The built-in id() function returns the unique identifier (memory address) of an object — an integer that stays constant during the object’s lifetime. In 2026 it remains the primary way to check object identity (is vs ==), debug reference issues, track object lifetimes, understand interning, and inspect memory behavior in CPython, especially useful in performance analysis, caching, and concurrency debugging. With Python 3.12–3.14+ bringing better memory management, free-threading support (object IDs remain unique per process), and improved introspection tools, id() is still invaluable for low-level understanding and debu...

Category: Built in Function • From: id() in Python 2026: Object Identity & Memory Address + Modern Introspection Use Cases

Q76. How does bool() in Python 2026: Truthy/Falsy Conversion + Modern Patterns & Use Cases work? Give a practical example.

bool() in Python 2026: Truthy/Falsy Conversion + Modern Patterns & Use Cases The built-in bool() function converts any value to a boolean (True or False) according to Python's truthy/falsy rules. It's one of the simplest yet most frequently used built-ins — powering every if-statement, while-loop, and logical operation behind the scenes. In 2026 it remains essential for input sanitization, default handling, conditional logic, data validation, and ML preprocessing. With Python 3.12–3.14+ offering better type hints, free-threading, and improved performance in conditional branches, bool() is still the cleanest way to explicitly convert values. This March 23, 2026 update explains truthy/falsy rules in det...

Category: Built in Function • From: bool() in Python 2026: Truthy/Falsy Conversion + Modern Patterns & Use Cases

Q77. What are the best practices for Bar plots in modern Python development?

Bar plots (bar charts) are one of the most effective and widely used visualizations for comparing categorical data — they clearly show differences in magnitude, rankings, trends over discrete groups, or composition within categories. They are perfect for business dashboards, survey results, performance comparisons, frequency counts, and any time you need to answer "how much" or "which is bigger" questions. In 2026, Python gives you excellent options: Matplotlib for full customization, Seaborn for beautiful statistical defaults, and Plotly for interactive, web-ready bars. Here's a practical guide with real examples you can copy and adapt. 1. Basic Setup & Sample Data import pandas as pd import matplotl...

Category: Data Manipulation • From: Bar plots

Q78. What are the best practices for Built-in function: range() in Python 2026 with Efficient Code in modern Python development?

Built-in function: range() in Python 2026 with Efficient Code The range() built-in is one of Python's most important tools for writing efficient loops and generating sequences. In 2026, understanding how to use range() properly remains essential for writing fast, memory-efficient, and clean code, especially in data processing, machine learning loops, and performance-critical applications. This March 15, 2026 update covers modern best practices, common pitfalls, and powerful patterns using range() in Python 2026. TL;DR — Key Takeaways 2026 range(stop) , range(start, stop) , range(start, stop, step) range() is memory-efficient — it generates numbers on demand (lazy) Use range() instead of...

Category: Efficient Code • From: Built-in function: range() in Python 2026 with Efficient Code

Q79. Explain 'Crawl in Python 2026: Building Modern Web Crawlers with Best Practices' in detail. Why is it important in 2026?

Crawl in Python 2026: Building Modern Web Crawlers with Best Practices Web crawling (also known as spidering) is the process of systematically browsing the internet to collect data. In 2026, building a

responsible and efficient crawler involves asynchronous I/O, respectful rate limiting, proper user-agent identification, robots.txt compliance, and clean data pipelines. This March 24, 2026 guide shows how to build a modern, classy Python crawler using current best practices with httpx, asyncio, BeautifulSoup, and ethical considerations. TL;DR — Key Takeaways 2026 Use asynchronous requests with httpx for speed Always respect robots.txt and add delays Implement proper error handling and logging ...

Category: Web Scrapping • From: Crawl in Python 2026: Building Modern Web Crawlers with Best Practices

Q80. Explain 'Working with Numpy arrays' in detail. Why is it important in 2026?

Working with NumPy arrays is foundational for numerical computing in Python — NumPy provides fast, memory-efficient multidimensional arrays (ndarrays) with vectorized operations, broadcasting, and a rich ecosystem for math, statistics, linear algebra, and more. In 2026, NumPy remains the backbone of scientific Python — powering pandas, Polars (via integration), scikit-learn, PyTorch, TensorFlow, Dask arrays, and large-scale data/ML workflows. Mastering NumPy array creation, indexing, reshaping, broadcasting, ufuncs, and aggregation lets you write concise, performant code that scales from small prototypes to massive datasets. Here’s a complete, practical guide to working with NumPy arrays: creation methods, i...

Category: Parallel Programming With Dask • From: Working with Numpy arrays

Q81. What are the best practices for For loop And List Comprehension in modern Python development?

For loop and list comprehension are two of the most essential tools for working with sequences in Python — both let you process items one by one, but they serve different purposes and have different strengths. The for loop is general-purpose, flexible, and readable for any kind of iteration — especially when logic is complex, involves side effects, or needs early exits. List comprehensions are a concise, expressive shortcut for creating new lists from existing iterables — ideal for simple transformations or filtering in a single line. In 2026, knowing when to use each (and when to combine them) is key to writing clean, efficient, Pythonic code. Here’s a complete, practical comparison and guide: syntax, use...

Category: Data Science Tool Box • From: For loop And List Comprehension

Q82. How does Iterating at once with asterisk work? Give a practical example.

Iterating at once with the asterisk (*) is one of Python’s most elegant and powerful features — it lets you unpack (or “splat”) the elements of an iterable in a single step. The asterisk operator (*) takes everything inside a list, tuple, set, or other iterable and spreads those elements out — either into function arguments, new collections, or unpacking assignments. It’s a clean, concise way to handle iterables without explicit loops in many common cases. In 2026, the * operator (often called “splat” or “unpack”) remains essential — used constantly for function calls, list/tuple/set creation, merging iterables, and unpacking assignments. Here’s a complete, practical guide to using * for iteration and unpack...

Category: Data Science Tool Box • From: Iterating at once with asterisk

Q83. How does Reading date and time data in Pandas work? Give a practical example.

Reading date and time data in Pandas is one of the most common and important tasks in data analysis — correctly parsing timestamps, dates, or time columns ensures you can perform time-based operations like resampling, shifting, grouping by periods, calculating durations, or plotting trends. Pandas provides powerful tools via pd.read_csv() , pd.to_datetime() , and the .dt accessor to read, parse, and manipulate datetime data efficiently. In 2026, mastering datetime parsing in pandas remains essential — especially with mixed formats, time zones, large files, or streaming data — and Polars offers even faster alternatives for massive datasets. Here’s a complete, practical guide to reading and working with da...

Category: Dates and Time • From: Reading date and time data in Pandas

Q84. Explain 'Multiple Parameters and Return Values' in detail. Why is it important in 2026?

Multiple parameters and return values are where Python functions become truly powerful. You can accept any number of inputs (positional, keyword, variable) and return one value, several values (via tuple unpacking), or complex structures (dicts, objects). In 2026, good functions use type hints, clear docstrings, and thoughtful signatures to make code intuitive and safe. Here’s a complete, practical guide to combining multiple parameters with flexible return patterns — with real-world examples and modern best practices. 1. Multiple Positional + Keyword Parameters def calculate_total(items: list[float], # positional required tax_rate: float = 0.08, # keyword or positional ...

Category: Data Science Tool Box • From: Multiple Parameters and Return Values

Q85. What are the best practices for Plot with Legend in modern Python development?

Adding a legend is one of the most important steps in making any multi-series plot readable and professional — it clearly labels each line, marker, bar, or area so viewers instantly understand what they're looking at without needing to guess. In 2026, legends are non-negotiable for EDA, presentations, dashboards, and publications. Here’s a practical guide showing how to add, position, style, and customize legends in Matplotlib (base control), Seaborn (beautiful defaults), and Plotly (interactive & shareable). 1. Basic Setup & Sample Data import pandas as pd import numpy as np import matplotlib.pyplot as plt import seaborn as sns import plotly.express as px # Realistic time-series data: two relat...

Category: Data Manipulation • From: Plot with Legend

Q86. Explain 'Repeated characters' in detail. Why is it important in 2026?

Repeated characters are one of the most powerful features in regular expressions — allowing you to match exact counts, ranges, or unbounded repetitions of a character, group, or pattern. In Python's re module, repetition is controlled by quantifiers: {n} for exactly n times, {m,n} for between m and n times, * for zero or more, + for one or more, and ? for zero or one. These quantifiers are greedy by default (match as much as possible) but can be made lazy with ? (e.g., *?) or possessive with + (rare). In 2026, mastering repetition quantifiers remains essential — used constantly in data validation (e.g., phone numbers, ZIP codes), text extraction, cleaning, parsing logs, and vectorized pandas/...

Category: Regular Expressions • From: Repeated characters

Q87. What are the best practices for The Future of Agentic AI – Trends and Predictions for 2027 in modern Python development?

As we stand in March 2026, the trajectory of Agentic AI points toward even greater autonomy, collaboration, and integration with real-world systems. Here are the major trends and predictions for 2027. Major Trends Expected in 2027 Enterprise Agentic Platforms : Companies will deploy fleets of specialized agents managed centrally Agent-to-Agent Economies : Agents will negotiate, trade services, and collaborate across organizations Multimodal Agents : Agents that seamlessly work with text, image, video, and audio Self-Improving Agents : Agents that can evaluate their own performance and improve over time Regulatory Frameworks : Governments will introduce specific rules for autonomous AI systems ...

Category: Agentic AI • From: The Future of Agentic AI – Trends and Predictions for 2027

Q88. Explain 'Summaries on multiple columns' in detail. Why is it important in 2026?

Summarizing multiple columns at once is one of the most powerful features of Pandas — and the .agg() method is the cleanest way to do it. Instead of calling .mean() , .median() , etc. separately on each column, .agg() lets you apply one or more functions across multiple columns in a single, readable call. In 2026, when datasets are larger and analysis pipelines are more complex, this method saves time and keeps your code maintainable. Here's a practical guide with real examples. 1. Basic Multi-Column Summary Apply the same set of functions to several columns — perfect for quick overviews. import pandas as pd data = { 'Name': ['John', 'Mary', 'Peter', 'Anna', 'Mike'], 'Age': [25, 32, ...

Category: Data Manipulation • From: Summaries on multiple columns

Q89. Explain 'iter() in Python 2026: Creating Iterators + Modern Patterns & Best Practices' in detail. Why is it important in 2026?

iter() in Python 2026: Creating Iterators + Modern Patterns & Best Practices The built-in iter() function returns an iterator object from an iterable — the foundation of every for-loop, generator expression, and lazy evaluation in Python. In 2026 it remains one of the most fundamental and frequently used built-ins,

powering list comprehensions, zip(), map(), filter(), enumerate(), async for, and custom iterator protocols. With Python 3.12–3.14+ offering faster iterator creation, improved type hinting for iterators (better generics), and free-threading compatibility for concurrent iteration, iter() is more performant and type-safe than ever. This March 23, 2026 update covers how iter() works today, real-wo...

Category: Built in Function • From: iter() in Python 2026: Creating Iterators + Modern Patterns & Best Practices

Q90. Explain 'Finding and replacing' in detail. Why is it important in 2026?

Finding and replacing substrings is one of the most essential string operations in Python — the replace() method finds all (or a limited number of) occurrences of a target substring and replaces them with a new string, returning the modified copy (strings are immutable). It’s fast, simple, and widely used for cleaning text, normalizing data, redacting sensitive info, templating, log processing, and preparing strings for parsing or display. In 2026, replace() remains a go-to tool — especially in pandas/Polars column cleaning, text preprocessing, configuration updates, and production pipelines where vectorized replacement scales to millions of rows efficiently. Here’s a complete, practical guide to finding...

Category: Regular Expressions • From: Finding and replacing

Q91. Explain 'Counting occurrences' in detail. Why is it important in 2026?

Counting occurrences of a substring, character, or pattern within a string is a fundamental text analysis task in Python — it helps quantify frequency, detect duplicates, validate data, measure density (e.g., keyword counts), and support preprocessing steps like feature extraction or anomaly detection. Python’s built-in count() method provides a fast, simple way to count non-overlapping occurrences of a substring, while re (regular expressions) enables overlapping counts, case-insensitive searches, and complex patterns. In 2026, counting remains essential — especially in data cleaning, NLP token frequency, log analysis, spam detection, and pandas/Polars string column summarization where vectorized .str.co...

Category: Regular Expressions • From: Counting occurrences

Q92. Explain 'Layering plots' in detail. Why is it important in 2026?

Layering plots (overlaying multiple visualizations on the same axes) is one of the most powerful techniques in data visualization — it lets you combine different representations of the same data (e.g., raw points + trend line + confidence band) or compare related variables (e.g., actual vs predicted) to reveal deeper insights that single plots cannot show. In 2026, layering is a core skill for exploratory data analysis, regression diagnostics, time-series decomposition, and professional reporting. Here’s a practical guide with real examples using Matplotlib (full control), Seaborn (statistical beauty), and Plotly (interactive layering). 1. Basic Setup & Sample Data import pandas as pd import numpy as ...

Category: Data Manipulation • From: Layering plots

Q93. Explain 'OR operator in re Module' in detail. Why is it important in 2026?

The OR operator in Python’s re module — represented by the vertical bar | — allows a regular expression to match either one pattern or another (or more) patterns. It’s the regex equivalent of logical OR, enabling flexible alternatives in matching — for example, matching different keywords, formats, or structures in text. In 2026, the OR operator remains one of the most essential regex features — used constantly in data validation (e.g., multiple date formats), text extraction (different log levels), search (synonyms or variants), and vectorized pandas/Polars string operations where pattern alternatives scale efficiently across large datasets. Here’s a complete, practical guide to the OR operator in Pytho...

Category: Regular Expressions • From: OR operator in re Module

Q94. How does Django 6.0 – Must-Know Features Released in 2025/2026 (Background Tasks, CSP & More) work? Give a practical example.

Django 6.0 (December 2025) brings production-ready background tasks, native Content Security Policy (CSP), template partials, modern email handling, and better async support — big steps toward secure, scalable, modern Django apps in 2026. No more Celery dependency for simple tasks, easier CSP hardening, and cleaner component-style templates. If you’re maintaining or starting Django projects in 2026, these features reduce third-party packages and improve security/performance out of the box. Here’s what matters most. 1. Built-in Tasks Framework – Background Work Without Celery/Redis Django now has a native way to define and run background tasks — perfect for emails, reports, cleanup jobs. # tasks.py f...

Category: Django • From: Django 6.0 – Must-Know Features Released in 2025/2026 (Background Tasks, CSP & More)

Q95. How does The re module work? Give a practical example.

The re module in Python is the built-in library for regular expressions (regex) — a powerful toolset for searching, matching, extracting, replacing, splitting, and validating patterns in text. Regex lets you describe complex text patterns (e.g., emails, dates, phone numbers, HTML tags, log formats) concisely and perform operations with high precision. In 2026, re remains essential — used constantly in data cleaning, log parsing, input validation, NLP preprocessing, web scraping, and pandas/Polars string column transformations where vectorized .str.extract() or .str.replace() scales to millions of rows efficiently. While simple string methods suffice for basic tasks, re unlocks pattern-based text proces...

Category: Regular Expressions • From: The re module

Q96. How does Web Development with Python in 2026 – FastAPI, Django & Flask Guide work? Give a practical example.

Web Development with Python in 2026 remains one of the strongest use-cases for the language. Python powers scalable APIs, full-featured web applications, admin panels, real-time services, and increasingly

AI-integrated backends. Three frameworks dominate the landscape right now: FastAPI – the modern choice for high-performance, async APIs (automatic OpenAPI docs, Pydantic validation, type hints everywhere) Django – still the go-to for full-stack applications, complex admin interfaces, ORM-heavy projects, and enterprise-grade security Flask – lightweight and flexible for microservices, prototypes, small-to-medium APIs, or when you want full control In 2026 benchmarks and real-world usage show: ...

Category: Web Development • From: Web Development with Python in 2026 – FastAPI, Django & Flask Guide

Q97. Explain 'Computing with Multidimensional Arrays' in detail. Why is it important in 2026?

Computing with Multidimensional Arrays forms the backbone of numerical and scientific computing in Python — enabling fast, vectorized operations on large, structured data like images, time series, climate grids, simulations, and ML tensors. NumPy provides the core ndarray for in-memory arrays, Dask extends it to out-of-core, parallel, and distributed computation for massive datasets, and xarray adds labeled dimensions, coordinates, and metadata for intuitive, netCDF-like workflows. In 2026, these tools remain essential — NumPy for speed on fit-in-memory data, Dask for scaling beyond RAM (terabytes+), and xarray for labeled, multidimensional analysis in geoscience, bioinformatics, and remote sensing. Masterin...

Category: Parallel Programming With Dask • From: Computing with Multidimensional Arrays

Q98. What are the best practices for Understanding now in Python's Datetime Module in modern Python development?

Understanding now in Python's Datetime Module is essential for capturing the current moment — the precise date and time when your code executes. The datetime.now() function returns a timezone-naive datetime object representing the local system time, serving as the foundation for timestamps, logging, scheduling, age calculations, time deltas, and real-time data processing. In 2026, Python’s datetime handling has matured: zoneinfo (stdlib since 3.9) provides robust timezone support, Polars offers ultra-fast columnar datetime operations, pandas remains the go-to for familiar workflows, and Pydantic integrates seamless datetime validation. This guide explores now() in depth: retrieving local/UTC/timezone-a...

Category: Datatypes • From: Understanding now in Python's Datetime Module

Q99. What are the best practices for Generator expressions in modern Python development?

Generator expressions are Python’s memory-efficient, lazy cousin of list comprehensions — they create a generator object that yields values one at a time instead of building a full list in memory. The syntax is almost identical — just use parentheses (...) instead of square brackets [...] — but the result is an iterator, not a list. This makes generator expressions ideal for large datasets, infinite sequences, streaming data, or when you only need to pass values to functions like sum() , max() , any() , or list() without storing the entire result. In 2026, generator expressions are a core tool for performance and scalability — especially in data

processing, file handling, API streaming, and working ...

Category: Data Science Tool Box • From: Generator expressions

Q100. What are the best practices for Numbered groups in modern Python development?

Numbered groups in Python’s re module are created with parentheses () — each opening parenthesis starts a numbered capturing group, beginning from 1 and counting left to right. These groups capture matched substrings for later use: extraction via .group(n) , .groups() , or findall() returning tuples; backreferences in patterns (\1 , \2); and reuse in replacements (\1 or \g<1>). In 2026, numbered groups remain a core regex feature — essential for extracting structured fields (names, dates, IDs), reformatting text, detecting duplicates, and vectorized pandas/Polars operations where capturing specific parts of matches scales efficiently across large datasets. Here’s a complete, practical guide to...

Category: Regular Expressions • From: Numbered groups