Python Interview Questions & Answers 2026

Random Set • March 24, 2026

Q1. Explain 'Iterating with a for loop' in detail. Why is it important in 2026?

Iterating with a for loop is the most natural and readable way in Python to step through every item in an iterable — a list, tuple, string, dictionary, set, range, file lines, or any object that supports iteration. The for loop automatically handles getting the next item until the iterable is exhausted — no manual indexing or counters needed. It’s clean, Pythonic, and used constantly for processing data, printing, transforming, filtering, aggregating, or building new collections. In 2026, the for loop remains the heart of iteration — from simple lists to massive datasets in pandas/Polars, file processing, API responses, and generators. Here’s a complete, practical guide to using for loops effectively: ...

Category: Data Science Tool Box • From: Iterating with a for loop

Q2. Explain 'memoryview with JAX in Python 2026: Zero-Copy NumPy → JAX Array Interop + Efficient ML Examples' in detail. Why is it important in 2026?

memoryview with JAX in Python 2026: Zero-Copy NumPy → JAX Array Interop + Efficient ML Examples JAX (with jax.numpy and jaxlib) has become one of the most popular numeric/ML frameworks in 2026 — especially for research, differentiable physics, and high-performance array computing on GPU/TPU. Combining memoryview with NumPy → JAX workflows allows true zero-copy slicing and interop for large arrays, avoiding expensive copies when preprocessing gigabyte-scale datasets, images, or scientific simulations. I've used this pattern in JAX-based diffusion models, PDE solvers, and large-scale time-series forecasting — slicing 4–12 GB arrays for batch augmentation or feature extraction without doubling host RAM befor...

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

Q3. How does Stacking arrays work? Give a practical example.

Stacking arrays is a fundamental operation for combining multiple arrays into a single higher-dimensional array — either along a new axis (stacking) or an existing one (concatenation). In NumPy and Dask, np.stack() / da.stack() adds a new axis, while np.concatenate() / da.concatenate() joins along an existing axis. In 2026, stacking remains essential for data preparation — combining time series channels, stacking image batches for ML, merging simulation runs, aligning multi-sensor readings, or building feature matrices — with Dask enabling scalable stacking of large/out-of-core arrays and xarray providing labeled stacking with dimension management. Here’s a complete, practical guide to stacking arrays in...

Category: Parallel Programming With Dask • From: Stacking arrays

Q4. What are the best practices for Index lookups in modern Python development?

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

Q5. Explain 'Using context managers' in detail. Why is it important in 2026?

Using context managers in Python is the idiomatic way to manage resources that need explicit setup and teardown — files, database connections, locks, network sockets, temporary directories, thread pools, and more — ensuring they are always properly released, even when exceptions occur. The with statement automatically calls __enter__() on entry and __exit__() on exit, handling cleanup reliably. In 2026, context managers remain a cornerstone of robust, exception-safe Python code — essential in data pipelines (file I/O, database transactions), web scraping (sessions, browsers), concurrency (locks), testing (mocks, patches), and production systems where resource leaks cause crashes, deadlocks, or memory exh...

Category: Writing Functions • From: Using context managers

Q6. How does Multiple statistics in pivot table work? Give a practical example.

Calculating multiple statistics in a pivot table is one of the most powerful features of pivot_table() in Pandas — it lets you summarize multiple columns with different aggregation functions (sum for sales, mean for profit, count for transactions, etc.) in a single, clean cross-tab view. This is ideal for financial reports, performance dashboards, cohort analysis, and any multi-metric breakdown. In 2026, pivot tables remain essential for turning raw data into actionable, multi-dimensional summaries. 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: Multiple statistics in pivot table

Q7. 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

Q8. What are the best practices for Set method symmetric difference in modern Python development?

Set method symmetric difference computes the set of elements that are in exactly one of the sets (but not both) — the symmetric difference, also known as the XOR of sets. It returns a new set containing items unique to each input, excluding any overlap. The symmetric_difference() method (or the ^ operator) is fast, readable, and perfect for finding differences, comparing collections, identifying unique items, or data reconciliation tasks like finding records present in one dataset but not the other (and vice versa). In 2026, symmetric difference remains a go-to operation — used constantly in data validation, deduplication, change detection, and production pipelines where you need to highlight exclusive ele...

Category: Efficient Code • From: Set method symmetric difference

Q9. How does Data Types For Data Science work? Give a practical example.

Data types are the foundation of data science in Python — they define how data is stored, accessed, manipulated, and analyzed. Choosing the right data type directly affects performance, memory usage, code clarity, and the accuracy of your results. In 2026, with datasets growing larger and more complex, understanding Python’s built-in types — and when to use specialized ones — is more important than ever. Here’s a practical overview of the data types most commonly used in data science workflows, with real examples and guidance on when to use each. 1. Numeric Types Numbers are the core of quantitative analysis. Python provides three main numeric types: int — whole numbers (unlimited size in Python 3) ...

Category: Datatypes • From: Data Types For Data Science

Q10. Explain 'Introduction to pandas DataFrame iteration' in detail. Why is it important in 2026?

Introduction to pandas DataFrame iteration is a key topic for anyone working with tabular data in Python — a DataFrame is pandas’ core 2D structure with labeled rows and columns, often holding mixed types (numbers, strings, dates, etc.). While it’s tempting to loop over rows or columns like a regular list, pandas is designed for vectorized operations that process entire columns/rows at once — often 10–100× faster than explicit loops. In 2026, mastering when to iterate (and when not to) is crucial for performance, especially with large datasets, data cleaning, feature engineering, or production pipelines. Iteration methods like iterrows() , itertuples() , and column access exist, but vectorization, apply, and...

Category: Efficient Code • From: Introduction to pandas DataFrame iteration

Q11. What are the best practices for Saving output in modern Python development?

Saving output of timeit lets you capture detailed timing results in a variable for further analysis, comparison, or logging — instead of just printing them to the console. In Jupyter notebooks, the -o option with %timeit (or %%timeit) stores everything in a TimeitResult object — mean, standard deviation, best/worst times, all runs, loops, and repeats — giving you full access to statistics for scripts, reports, visualizations, or automated benchmarks. In 2026, saving timeit output is standard for reproducible performance testing, regression detection in CI/CD, comparing implementations, and tracking speed over time in production notebooks. Here’s a complete, practical guide to saving and using timeit ...

Category: Efficient Code • From: Saving output

Q12. How does Deferring computation with `delayed` work? Give a practical example.

Deferring computation with delayed is Dask’s low-level, flexible way to build lazy computation graphs one function call at a time — perfect when you need fine-grained control over parallelism, want to delay individual operations (not just DataFrame/Array methods), or are composing custom Python functions into scalable pipelines. Unlike high-level Dask collections (dd.read_csv , da.from_array), dask.delayed wraps any Python callable — turning normal function calls into Delayed objects that record dependencies without executing until .compute() . In 2026, delayed remains essential for custom workflows, non-DataFrame computations, integrating legacy code, building dynamic pipelines, or optimizing task ...

Category: Parallel Programming With Dask • From: Deferring computation with `delayed`

Q13. 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

Q14. How does Building delayed pipelines work? Give a practical example.

Building delayed pipelines is one of the most flexible and powerful patterns in Dask — it lets you compose complex, multi-step workflows as a directed acyclic graph (DAG) of lazy computations using the @delayed decorator or dask.delayed wrapper. Each step is deferred until you call dask.compute() , allowing Dask to automatically parallelize, optimize, and schedule the entire pipeline across multiple cores, processes, or distributed clusters. In 2026, delayed pipelines are foundational for custom ETL, feature engineering, simulation ensembles, machine learning preprocessing, and scientific analysis — especially when high-level collections (DataFrame/Array) are too rigid or you need to integrate arbitrary P...

Category: Parallel Programming With Dask • From: Building delayed pipelines

Q15. What are the best practices for Group by to pivot table in modern Python development?

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

Q16. What are the best practices for Using json module in modern Python development?

Using json module is fundamental for working with JSON data in Python — the built-in json library provides simple, reliable methods to serialize (encode) Python objects to JSON strings/files and deserialize (decode) JSON back into Python data structures (dicts, lists, etc.). In 2026, json remains the standard for API responses, configuration files, data exchange, logs, earthquake metadata, web scraping, and NoSQL exports — fast enough for most use cases, human-readable, and universally compatible. For higher performance on large JSON/JSONL files, pair with orjson or ujson ; for parallel processing, use Dask Bags; for tabular JSON, use pandas/Polars; for labeled multi-dimensional JSON-derived data, use x...

Category: Parallel Programming With Dask • From: Using json module

Q17. Explain 'Introduction Datatypes' in detail. Why is it important in 2026?

Introduction to Datatypes in Python Datatypes are the foundation of any programming language — they define what kind of data can be stored, how it behaves, and what operations are possible on it. Python is dynamically typed, meaning you don’t declare types explicitly; the interpreter infers them at runtime. This flexibility makes Python fast to write and read, but understanding datatypes deeply is key to writing efficient, bug-free, and performant code. In 2026, Python’s core datatypes remain largely the same as in

earlier versions, but modern usage patterns have evolved significantly with tools like Polars, Dask, NumPy, Pydantic, dataclasses, and type hints (mypy, pyright). This guide covers all major buil...

Category: Datatypes • From: Introduction Datatypes

Q18. How does Functions as variables work? Give a practical example.

Functions as variables (or functions as first-class objects) is one of Python's most elegant and powerful features — functions are treated like any other object (int, str, list, dict, etc.), so you can assign them to variables, pass them as arguments, return them from other functions, store them in data structures, and use them dynamically. This enables higher-order functions, callbacks, decorators, strategy patterns, functional programming techniques, plugin systems, and code that adapts at runtime. In 2026, this capability remains central to clean, flexible, reusable Python code — powering decorators (logging, timing, caching), callbacks in GUIs/async/ML pipelines, dynamic dispatch in pandas/Polars transform...

Category: Writing Functions • From: Functions as variables

Q19. How does Advanced Caching Strategies for Agentic AI Systems in 2026 work? Give a practical example.

Caching is one of the most powerful and underutilized tools for reducing cost and latency in Agentic AI systems. In 2026, advanced caching strategies can cut operational costs by 40–70% while significantly improving response times. This guide covers advanced caching techniques specifically designed for multi-agent systems built with CrewAI, LangGraph, and LlamaIndex as of March 24, 2026. Why Advanced Caching is Essential for Agentic AI Agentic systems are naturally cache-friendly because they often repeat similar reasoning patterns, tool calls, and retrieval operations. Without intelligent caching, the same work is repeated unnecessarily, driving up both cost and latency. Types of Caching in Agentic AI ...

Category: Agentic AI • From: Advanced Caching Strategies for Agentic AI Systems in 2026

Q20. How does Stripping characters work? Give a practical example.

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

Q21. How does Timing array computations work? Give a practical example.

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

Q22. Explain 'collections.Counter()' in detail. Why is it important in 2026?

collections.Counter() is Python’s built-in, high-performance way to count occurrences of hashable elements in any iterable — lists, strings, tuples, files, API results, or even other Counters. It returns a dict-like object where keys are the elements and values are their counts, with a rich API for arithmetic, most common items, and more. In 2026, Counter remains essential — faster and more readable than manual dict counting or loops, and perfect for frequency analysis, data cleaning, statistics, text processing, and production pipelines handling large datasets. Here’s a complete, practical guide to using collections.Counter: basic counting, accessing results, advanced features, real-world patterns, and mode...

Category: Efficient Code • From: collections.Counter()

Q23. 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

Q24. What are the best practices for ISO 8601 format with Exmples in modern Python development?

ISO 8601 format with examples is the international standard for representing dates, times, durations, and intervals in a clear, unambiguous, and machine-readable way. Widely adopted in computing, APIs, databases, JSON, XML, and data interchange, ISO 8601 eliminates confusion from regional formats (MM/DD/YYYY vs DD/MM/YYYY) and ensures consistent sorting, parsing, and comparison. In 2026, ISO

8601 remains the gold standard — used everywhere from timestamps in logs and databases to calendar APIs, financial transactions, scientific data, and modern web services. Mastering it is essential for reliable, interoperable code. Here’s a complete, practical guide to ISO 8601: core formats for dates, times, datetimes, t...

Category: Dates and Time • From: ISO 8601 format with Exmples

Q25. Explain 'The yield keyword' in detail. Why is it important in 2026?

The yield keyword is Python’s core mechanism for creating generators — special iterable functions that produce values one at a time, lazily, without loading the entire sequence into memory at once. When a function contains yield , calling it returns a generator object instead of executing the body immediately. Each call to next() (or iteration in a for loop) resumes execution from the last yield , runs until the next yield (or end), returns the yielded value, and pauses again — preserving local state between yields. In 2026, yield remains essential — powering memory-efficient data processing, infinite sequences, coroutines (pre-async/await era), custom iterators, and streaming pipelines in pandas/Polar...

Category: Writing Functions • From: The yield keyword

Q26. How does import() in Python 2026: How It Works, Security Risks & Modern Alternatives (importlib) work? Give a practical example.

__import__() in Python 2026: How It Works, Security Risks & Modern Alternatives (importlib) The __import__() built-in function is one of Python's most powerful — and most dangerous — tools for dynamic module loading. In 2026, even though it's still part of the language, almost every style guide, security audit, and modern codebase recommends avoiding it in favor of importlib.import_module() or importlib.util helpers. Why? Because __import__ is low-level, hard to secure, and error-prone — especially in plugins, configuration-driven systems, or code that loads modules from untrusted sources. I've seen __import__ cause real damage in production: remote code execution in plugin systems, import bombs, and ...

Category: built in function • From: import() in Python 2026: How It Works, Security Risks & Modern Alternatives (importlib)

Q27. What are the best practices for The .agg() method in modern Python development?

The .agg() method is one of the most powerful and flexible tools in Pandas for summarizing data — especially when you need to apply multiple aggregation functions at once, or different functions to different columns. In 2026, it's still the go-to method for clean, expressive groupby aggregations and quick DataFrame-wide summaries. Whether you're calculating sales totals, user metrics, sensor statistics, or financial summaries, .agg() lets you do it all in one readable line. Basic Usage: Multiple Aggregations on Selected Columns Use .agg() directly on a DataFrame or after a groupby() to apply several functions at once. import pandas as pd data = { 'Name': ['John', 'Mary', 'Peter', 'Anna', 'Mik...

Category: Data Manipulation • From: The .agg() method

Q28. Explain 'Using enumerate()' in detail. Why is it important in 2026?

Iterating with enumerate() is one of Python’s most useful and frequently used iteration tools — it lets you loop over any iterable while automatically getting both the element and its index (position) at the same time. Instead of manually managing a counter variable, enumerate() pairs each item with a number — starting at 0 by default, or any value you choose with start= . This makes it perfect for numbered lists, tracking positions, building indexed data, or when order matters in processing. In 2026, enumerate() remains a core Pythonic pattern — used everywhere from simple lists to pandas DataFrames, file line numbering, and custom indexing. Here’s a complete, practical guide to using enumerate() : ho...

Category: Data Science Tool Box • From: Using enumerate()

Q29. Explain 'Analyzing Earthquake Data' in detail. Why is it important in 2026?

Analyzing Earthquake Data is a classic real-world task in data science and geophysics — exploring seismic events to understand patterns, magnitudes, locations, depths, and risks. The USGS provides open, real-time earthquake catalogs via their FDSN web services, downloadable as CSV, making it easy to fetch, process, and visualize recent or historical data. In 2026, this workflow is standard for students, researchers, journalists, and disaster analysts — using pandas/Dask for loading and aggregation, matplotlib/seaborn/hvplot for visualization, and geospatial tools (geopandas, cartopy, folium) for mapping epicenters. With Dask, you can scale to years of global data without memory limits; with Polars, you get bla...

Category: Parallel Programming With Dask • From: Analyzing Earthquake Data

Q30. How does Loading datetimes with parse_dates work? Give a practical example.

Loading datetimes with parse_dates is one of pandas’ most powerful and time-saving features — it automatically converts date/time columns from strings to proper datetime64 objects during import, so you can immediately use vectorized .dt accessors, resampling, time zone conversions, or period grouping without extra steps. In 2026, correct datetime parsing on load remains critical — especially with mixed formats, time zones, large files, or streaming data — and Polars offers even faster, more memory-efficient alternatives for massive datasets. Getting this right avoids slow post-import conversions, parsing errors, and invalid timestamps that break downstream analysis. Here’s a complete, practical guide to ...

Category: Dates and Time • From: Loading datetimes with parse_dates

Q31. What are the best practices for Plotting missing values in modern Python development?

Plotting missing values is one of the fastest ways to move from “there are missings” to “where, how many, and in what pattern?” Numbers alone (counts, percentages) hide clustering, correlations, and group-specific gaps. Visuals reveal the story — whether missingness is random, systematic, or tied to specific variables/groups — guiding your imputation or drop strategy. In 2026, the gold standard remains missingno for quick, beautiful diagnostics, combined with seaborn heatmaps for deeper pattern analysis. Here’s a complete, practical guide with real examples and best practices. 1. Install & Quick Setup # Install once (if needed) # !pip install missingno import pandas as pd import missingno as msn...

Category: Data Manipulation • From: Plotting missing values

Q32. What are the best practices for From String to datetime in modern Python development?

Converting strings to datetime objects is one of the most frequent tasks in Python programming — whether you're processing logs, API responses, CSV files, user input, or database timestamps. Python’s datetime module makes this reliable and flexible, but getting the format right is key to avoiding frustrating errors. Here’s a practical, step-by-step guide to mastering string-to-datetime conversion in 2026 — with examples, common formats, best practices, and modern alternatives. Basic Conversion with strptime() Use datetime.strptime() — it parses a string according to a format string you provide. from datetime import datetime date_str = "2026-03-15 14:30:00" dt = datetime.strptime(date_str, "%Y...

Category: Data Manipulation • From: From String to datetime

Q33. What are the best practices for Passing invalid arguments in modern Python development?

Passing invalid arguments to a function is one of the most frequent causes of runtime errors in Python. When the type, value, count, or format of an argument doesn’t match what the function expects, Python raises clear exceptions — usually TypeError (wrong type), ValueError (bad value), or IndexError / KeyError (missing element). In production code, these can crash your program or produce silent bugs if unhandled. In 2026, robust functions detect invalid arguments early, raise meaningful errors, and provide helpful feedback — turning crashes into controlled, debuggable failures. Here’s a complete, practical guide to what happens when invalid arguments are passed, how to catch and handle them gracefully...

Category: Data Science Tool Box • From: Passing invalid arguments

Q34. What are the best practices for All parts of Pandas in modern Python development?

All parts of Pandas make it the go-to library for data manipulation, cleaning, analysis, and preparation in Python — a fast, flexible, and expressive toolkit built on NumPy that handles structured data like spreadsheets or SQL tables. In 2026, Pandas remains foundational for data science, machine learning pipelines, business intelligence, scientific research, and production workflows — with its core structures

(Series, DataFrame, Index), powerful grouping/resampling, merging/joining, time-series tools, I/O capabilities, and seamless integration with visualization (Matplotlib/Seaborn), statistical libraries, and big-data alternatives like Polars. Mastering these components lets you read, transform, aggregate, a...

Category: Dates and Time • From: All parts of Pandas

Q35. How does Rebrowser vs Camoufox Comparison 2026 – Which is Better for Playwright Stealth in Python Web Scrapping? work? Give a practical example.

In 2026, when doing web scrapping with Playwright, basic stealth plugins like playwright-stealth are often not enough against advanced anti-bot systems (Cloudflare Turnstile, DataDome, PerimeterX, Akamai). This pushes many Python developers toward stronger solutions: Rebrowser Playwright and Camoufox . Both tools aim to make Playwright automation nearly undetectable — but they use very different approaches. This 2026 comparison shows which one is better for your web scrapping project. Quick Comparison Table – Rebrowser vs Camoufox (March 2026) Aspect Rebrowser Playwright Camoufox Winner Browser Engine Chromium (patched Playwright) Firefox (custom C++ build) — Detection Evasion...

Category: Web Scrapping • From: Rebrowser vs Camoufox Comparison 2026 – Which is Better for Playwright Stealth in Python Web Scrapping?

Q36. Explain 'id() in Python 2026: Object Identity & Memory Address + Modern Introspection Use Cases' in detail. Why is it important in 2026?

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

Q37. Explain 'Python Programming Language in 2026 – Complete Guide & Why It Still Dominates' in detail. Why is it important in 2026?

Python remains the most popular and versatile programming language in 2026. From beginners to large enterprises, Python continues to dominate web development, data science, artificial intelligence, automation, and scientific computing. Its clean syntax, massive ecosystem, and strong community make it the go-to language for both rapid prototyping and production systems. This updated 2026 guide covers why Python is still the best choice and what has changed in the past year. Why Python Dominates in 2026 AI & Machine Learning Leader : PyTorch 2.5+, TensorFlow, JAX, and Hugging Face are all Python-first Data

Science & Analytics : Polars has overtaken pandas in many performance-critical workflows while pa...

Category: Introduction • From: Python Programming Language in 2026 – Complete Guide & Why It Still Dominates

Q38. Explain 'Backreferences' in detail. Why is it important in 2026?

Backreferences in Python's re module let you refer back to previously captured groups within the same pattern or replacement string — using \1 , \2 , etc. (for numbered groups) or (?P=name) (for named groups). They are incredibly useful for matching repeated substrings (e.g., duplicate words, paired tags, quoted strings, or mirrored patterns), enforcing consistency, or reusing captured text in replacements (e.g., reformatting, swapping parts). In 2026, backreferences remain a key regex feature — essential in data validation, text normalization, log parsing, HTML/XML tag matching, and vectorized pandas/Polars string operations where detecting or transforming repeated patterns scales efficiently across lar...

Category: Regular Expressions • From: Backreferences

Q39. How does Stacking two-dimensional arrays work? Give a practical example.

Stacking two-dimensional arrays is a common and essential operation when combining multiple 2D arrays (matrices, images, feature maps, time series grids) into higher-dimensional structures — either along an existing axis (concatenation) or a new one (stacking). In NumPy, np.concatenate() joins along existing axes, while np.stack() adds a new axis; np.vstack() , np.hstack() , and np.dstack() are convenience wrappers for vertical, horizontal, and depth-wise stacking. Dask provides equivalent lazy, parallel versions (da.concatenate() , da.stack() , da.vstack() , da.hstack() , da.dstack()). In 2026, correct stacking is critical for data preparation — merging multi-sensor images, stacking time windows...

Category: Parallel Programming With Dask • From: Stacking two-dimensional arrays

Q40. How does Anonymous functions work? Give a practical example.

Anonymous functions — better known as lambda functions — are Python's way of creating small, nameless, inline functions for one-off use. Defined with the lambda keyword, they take any number of arguments but contain only a single expression — no statements, no multi-line logic, no docstrings. They're perfect when you need a tiny function as an argument to map() , filter() , sorted() , or callbacks — without cluttering your code with a full def . In 2026, lambdas remain a core Pythonic tool — concise, functional, and elegant when used right. Here's a complete, practical guide to writing, using, and mastering anonymous functions — with modern patterns, type hints, and when to prefer regular functions i...

Category: Data Science Tool Box • From: Anonymous functions

Q41. How does callable() in Python 2026: Check If Object Is Callable + Modern Patterns & Use Cases work? Give a practical example.

callable() in Python 2026: Check If Object Is Callable + Modern Patterns & Use Cases The built-in callable() function returns True if the object appears callable (can be called with parentheses), False otherwise. In 2026 it remains a lightweight, essential tool for introspection, dynamic dispatch, type guards, plugin systems, dependency injection, and defensive programming — especially in frameworks like FastAPI, Pydantic, and ML pipelines where you often need to check if something is a function, method, class, or callable object before invoking it. With Python 3.12–3.14+ bringing improved type hinting (Callable support), free-threading compatibility, and better introspection performance, callable() i...

Category: Built in Function • From: callable() in Python 2026: Check If Object Is Callable + Modern Patterns & Use Cases

Q42. What are the best practices for Adjusting cases in modern Python development?

Adjusting cases is a fundamental string operation in Python — converting text to uppercase, lowercase, title case, or capitalizing the first letter helps standardize data, improve readability, prepare text for comparison/search, clean user input, format reports, or normalize columns in data analysis. Python's built-in string methods (upper() , lower() , title() , capitalize() , casefold() , swapcase()) are fast, immutable (return new strings), and vectorized in pandas/Polars for large-scale text processing. In 2026, case adjustment remains essential — especially for text cleaning in NLP, data preprocessing, database normalization, log parsing, and user-facing displays where consistent casing improves ma...

Category: Regular Expressions • From: Adjusting cases

Q43. How does property() in Python 2026: Properties, Getters/Setters & Modern Patterns work? Give a practical example.

property() in Python 2026: Properties, Getters/Setters & Modern Patterns The built-in property() function turns a method into a "property" — allowing getter, setter, and deleter access using dot notation like a regular attribute. In 2026 it remains the most Pythonic and widely used way to implement computed/read-only attributes, validation on assignment, lazy evaluation, and clean encapsulation without exposing implementation details. With Python 3.12–3.14+ improving property performance (faster descriptor lookup), better type hinting support for properties, and free-threading compatibility for concurrent property access (when used safely), property() is more efficient and type-safe than ever. This March ...

Category: Built in Function • From: property() in Python 2026: Properties, Getters/Setters & Modern Patterns

Q44. What are the best practices for Detecting any missing values with .isna().any() in modern Python development?

Detecting any missing values with .isna().any() is the fastest, most lightweight way to answer the question: “Does this column (or the whole dataset) have even a single missing value?” In real workflows, you run this check seconds after loading data — before diving into counts, heatmaps, or imputation strategies. In pandas, .isna().any() returns a boolean Series (or single boolean for scalars) — True if the column contains at least one NaN/None/null, False otherwise. It’s blazing fast and memory-efficient, especially on huge DataFrames. 1. Basic Usage (Pandas) import pandas as pd # Realistic example: messy survey data data = { 'name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'], 'ag...

Category: Data Manipulation • From: Detecting any missing values with .isna().any()

Q45. Explain 'eval() in Python 2026: Dynamic Expression Evaluation + Security Risks & Safe Alternatives' in detail. Why is it important in 2026?

eval() in Python 2026: Dynamic Expression Evaluation + Security Risks & Safe Alternatives The built-in eval() function evaluates a string as a Python expression and returns the result. In 2026 it remains one of the most powerful — and most dangerous — built-ins, used for dynamic calculation, configuration parsing, interactive shells, and some metaprogramming scenarios. However, its ability to execute arbitrary code makes it a top security risk when handling untrusted input. With Python 3.12–3.14+ offering better AST parsing, improved restricted execution tools, and growing security best practices (restrictedpython, ast.literal_eval, sandboxing), eval() is used far more cautiously. This March 23, 2026 upda...

Category: Built in Function • From: eval() in Python 2026: Dynamic Expression Evaluation + Security Risks & Safe Alternatives

Q46. Explain 'memoryview with NumPy in Python 2026: Zero-Copy Views, Efficient Slicing & Real ML Examples' in detail. Why is it important in 2026?

memoryview with NumPy in Python 2026: Zero-Copy Views, Efficient Slicing & Real ML Examples NumPy arrays and memoryview are a perfect match in 2026 — both support the buffer protocol, allowing you to create zero-copy, high-performance views into large numerical arrays without duplicating memory. This is especially powerful for machine learning preprocessing, image/video handling, scientific simulations, and any workflow involving gigabyte-scale arrays where copying would kill performance or exceed RAM. I’ve used memoryview + NumPy extensively in computer vision pipelines, time-series feature extraction, and large-scale data augmentation — slicing 4 GB image batches in microseconds without extra allocation...

Category: built in function • From: memoryview with NumPy in Python 2026: Zero-Copy Views, Efficient Slicing & Real ML Examples

Q47. What are the best practices for CrewAI vs LangGraph vs AutoGen 2026 – Which Framework Should You Use? in modern Python development?

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?

Q48. What are the best practices for float() in Python 2026: Floating-Point Number Creation + Modern Precision & Use Cases in modern Python development?

float() in Python 2026: Floating-Point Number Creation + Modern Precision & Use Cases The built-in float() function converts a number or string to a floating-point number (IEEE 754 double precision). In 2026 it remains the primary way to create floats from integers, strings, or other numeric types — essential for scientific computing, data processing, machine learning (loss scaling, normalization), financial calculations, and graphics/physics simulations. With Python 3.12–3.14+ offering faster float operations, better decimal interop, free-threading compatibility for concurrent numeric code, and growing use of float32/float16 in ML frameworks (PyTorch, JAX), float() is more versatile than ever. This March...

Category: Built in Function • From: float() in Python 2026: Floating-Point Number Creation + Modern Precision & Use Cases

Q49. Explain 'Extracting Dask array from HDF5' in detail. Why is it important in 2026?

Extracting Dask array from HDF5 is a key technique for scalable, out-of-core processing of massive scientific datasets stored in HDF5 files — especially when the data exceeds RAM limits or requires parallel/distributed computation. HDF5's native chunking and partial I/O pair perfectly with Dask arrays: dask.array.from_array() wraps the HDF5 dataset into a lazy, chunked Dask array that mirrors the file's chunk layout, enabling Dask to perform computations (mean, sum, filtering, ML preprocessing) in parallel without loading the full dataset. In 2026, this pattern is standard in climate science, genomics, satellite imagery, particle physics, and high-performance simulation analysis — integrating h5py for file a...

Category: Parallel Programming With Dask • From: Extracting Dask array from HDF5

Q50. What are the best practices for sum() in Python 2026: Summing Iterables + Modern Numeric Patterns & Best Practices in modern Python development?

sum() in Python 2026: Summing Iterables + Modern Numeric Patterns & Best Practices The built-in sum() function computes the sum of an iterable of numbers (with optional start value). In 2026 it remains one of the most frequently used built-ins for aggregation, statistical calculations, loss averaging in ML, financial totals, time-series integration, and any scenario requiring fast summation of numeric sequences. With Python 3.12–3.14+ improving numeric performance (faster summation loops), better type hinting for

numeric iterables, and free-threading compatibility for concurrent summation (when used safely), sum() is more efficient and type-safe than ever. This March 24, 2026 update covers how sum() works...

Category: Built in Function • From: sum() in Python 2026: Summing Iterables + Modern Numeric Patterns & Best Practices

Q51. Explain 'object() in Python 2026: Base Class & Minimal Instance Creation + Modern Use Cases' in detail. Why is it important in 2026?

object() in Python 2026: Base Class & Minimal Instance Creation + Modern Use Cases The built-in object() is the most fundamental class in Python — every class inherits from it by default if no other base class is specified. Calling object() creates a plain, empty instance with no attributes or methods beyond those defined in object itself. In 2026 it remains the go-to base class for minimal objects, sentinel values, type annotations, metaprogramming, and as a safe starting point when you want no inherited behavior except the basics (__str__, __repr__, __eq__, etc.). With Python 3.12–3.14+ improving object creation speed, enhancing free-threading safety for simple instances, and better type system supp...

Category: Built in Function • From: object() in Python 2026: Base Class & Minimal Instance Creation + Modern Use Cases

Q52. How does Adding and Extending Python Dictionaries: Flexible Data Manipulation work? Give a practical example.

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

Q53. Explain 'Humanizing Differences: Making Time Intervals More Readable with Pendulum' in detail. Why is it important in 2026?

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

Q54. What are the best practices for Supported metacharacters in modern Python development?

Supported metacharacters are the special symbols in Python's regular expressions (via the re module) that have predefined meanings beyond their literal character — they define patterns, repetition, positions, groups, character classes, and more. These metacharacters make regex powerful for matching complex text structures like emails, dates, phone numbers, log formats, HTML tags, or custom delimiters. In 2026, understanding metacharacters remains fundamental — they are used constantly in data validation, text extraction, cleaning, parsing, and vectorized operations in pandas/Polars string columns. Knowing when and how to escape them (with \ ) prevents syntax errors and unintended matches. Here's a complet...

Category: Regular Expressions • From: Supported metacharacters

Q55. How does memoryview with NumPy & PyTorch in Python 2026: Zero-Copy Views, Efficient Slicing & ML Interop Examples work? Give a practical example.

memoryview with NumPy & PyTorch in Python 2026: Zero-Copy Views, Efficient Slicing & ML Interop Examples Combining memoryview with NumPy and PyTorch unlocks extremely efficient, zero-copy workflows in 2026 — especially when moving large arrays/tensors between preprocessing (NumPy), model input (PyTorch), and visualization/analysis. By using memoryview, you avoid expensive copies when slicing gigabyte-scale image batches, feature matrices, or time-series data, which is critical for memory-constrained GPUs or large-scale training. I've used this pattern heavily in computer vision pipelines (image augmentation), time-series forecasting, and transfer learning setups — passing 4–10 GB datasets to PyTorch DataL...

Category: built in function • From: memoryview with NumPy & PyTorch in Python 2026: Zero-Copy Views, Efficient Slicing & ML Interop Examples

Q56. 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

Q57. What are the best practices for Errors and exceptions in modern Python development?

Errors and exceptions are two closely related but distinct concepts in Python. An error is any problem that prevents code from running correctly — from syntax mistakes (caught before execution) to runtime issues (during execution). Exceptions are a specific type of runtime error: special objects Python raises when an abnormal condition occurs, such as dividing by zero, accessing a missing key, or file not found. In 2026, understanding and handling exceptions is essential for writing robust, production-ready code that fails gracefully instead of crashing. Here's a complete, practical introduction to errors and exceptions: how they work, the most common types, how to catch and handle them, and modern best prac...

Category: Data Science Tool Box • From: Errors and exceptions

Q58. What are the best practices for staticmethod() in Python 2026: Static Methods, Utility Functions & Modern Best Practices in modern Python development?

staticmethod() in Python 2026: Static Methods, Utility Functions & Modern Best Practices The built-in staticmethod() decorator turns a method into a static method — one that receives neither the instance (self) nor the class (cls) as an implicit first argument. In 2026 it remains the standard way to define utility functions, helper methods, or class-level operations that logically belong to a class but do not depend on instance or class state. With Python 3.12–3.14+ improving method lookup performance, better type hinting support for static methods, and free-threading compatibility for concurrent static calls, staticmethod is more efficient and type-safe than ever. This March 24, 2026 update explains ...

Category: Built in Function • From: staticmethod() in Python 2026: Static Methods, Utility Functions & Modern Best Practices

Q59. How does setattr() in Python 2026: Dynamic Attribute Setting + Modern Patterns & Safety work? Give a practical example.

setattr() in Python 2026: Dynamic Attribute Setting + Modern Patterns & Safety The built-in setattr(obj, name, value) function dynamically sets an attribute on an object by name — the counterpart to obj.name = value . In 2026 it remains a key tool for metaprogramming, configuration injection, plugin systems, dependency injection (FastAPI, Pydantic), testing/mocking, and dynamic object modification where attribute names are computed or come from external sources (configs, APIs, user input). With Python 3.12–3.14+ improving attribute setting performance, enhancing type hinting for dynamic access, and free-threading support for concurrent object modification (with locks when needed), setattr() is more relia...

Category: Built in Function • From: setattr() in Python 2026: Dynamic Attribute Setting + Modern Patterns & Safety

Q60. What are the best practices for isinstance() in Python 2026: Type Checking + Modern Patterns & Best Practices in modern Python development?

isinstance() in Python 2026: Type Checking + Modern Patterns & Best Practices The built-in isinstance(obj, class_or_tuple) function checks whether an object is an instance or subclass of a specified class (or tuple of classes). In 2026 it remains the recommended, safe, and idiomatic way to perform runtime type checking — essential for defensive programming, input validation, polymorphism handling, data processing, ML pipelines, and frameworks like FastAPI, Pydantic, and JAX/NumPy. With Python 3.12–3.14+ improving type hinting (better generics, Self, TypeGuard), free-threading support for concurrent checks, and growing use in type-safe dynamic code, isinstance() is more powerful and type-aware than ever. T...

Category: Built in Function • From: isinstance() in Python 2026: Type Checking + Modern Patterns & Best Practices

Q61. Explain 'len() in Python 2026: Length of Sequences & Modern Patterns & Best Practices' in detail. Why is it important in 2026?

len() in Python 2026: Length of Sequences & Modern Patterns & Best Practices The built-in len() function returns the number of items in a container (list, tuple, string, dict, set, etc.) or the length of user-defined objects that implement __len__() . In 2026 it remains one of the most frequently called built-ins — essential for bounds checking, pagination, data validation, loop control, memory estimation, and ML batch sizing. With Python 3.12–3.14+ improving container performance, free-threading support for concurrent length queries, and better type hinting for sized objects, len() is faster and safer in modern code. This March 23, 2026 update covers how len() behaves today, real-world patterns, perform...

Category: Built in Function • From: len() in Python 2026: Length of Sequences & Modern Patterns & Best Practices

Q62. What are the best practices for repr() in Python 2026: Official String Representation + Modern Debugging & Serialization Use Cases in modern Python development?

getattr() in Python 2026: Dynamic Attribute Access + Modern Patterns & Safety The built-in getattr(obj, name, default=None) function dynamically retrieves an attribute from an object by name — the safe, flexible counterpart to obj.name . In 2026 it remains a cornerstone of metaprogramming, plugin systems, configuration-driven code, dependency injection (FastAPI, Pydantic), testing/mocking, and dynamic dispatch where attribute names are determined at runtime. With Python 3.12–3.14+ improving attribute lookup speed, enhancing type hinting for dynamic access, and free-threading support for concurrent object inspection, getattr() is more reliable and performant than ever. This March 23, 2026 update explains ...

Category: Built in Function • From: repr() in Python 2026: Official String Representation + Modern Debugging & Serialization Use Cases

Q63. 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

Q64. What are the best practices for Renaming decorated functions in modern Python development?

Renaming decorated functions is a crucial best practice when using decorators in Python — without proper care, the wrapped function loses its original identity (__name__ , docstring, type hints, annotations), causing confusion during debugging, profiling, logging, or introspection (e.g., print(func.__name__) shows wrapper instead of the real name). The @functools.wraps decorator solves most of this by copying metadata from the original function to the wrapper, while manual renaming of __name__ or other attributes adds extra clarity (e.g., decorated_my_func). In 2026, metadata preservation is non-negotiable — modern tools (mypy, IDEs, logging, monitoring) rely on accurate function names and signature...

Category: Parallel Programming With Dask • From: Renaming decorated functions

Q65. What are the best practices for Adjusting timezone vs changing tzinfo in modern Python development?

Adjusting timezone vs changing tzinfo is one of the most important distinctions when working with time zones in Python — it determines whether you get correct time math or silent bugs. The astimezone() method properly converts a timezone-aware datetime to another time zone by adjusting the clock time to represent the same moment (accounting for offset and DST changes). In contrast, using replace(tzinfo=...) simply attaches or swaps the tzinfo object without changing the local time — which almost always produces incorrect results because the same numbers now represent a different moment. In 2026, understanding this difference is critical — incorrect timezone handling causes bugs in logs, scheduling, finan...

Category: Dates and Time • From: Adjusting timezone vs changing tzinfo

Q66. How does Setting up a Selector in Python 2026: Best Practices for Web Scraping work? Give a practical example.

Setting up a Selector in Python 2026: Best Practices for Web Scraping Setting up a proper selector is the first and most critical step in any web scraping project. In 2026, with modern async tools and dynamic websites, choosing the right way to create and manage selectors can dramatically improve speed, reliability, and maintainability of your scraper. This March 24, 2026 guide covers the best ways to set up

selectors using BeautifulSoup, parsel, and Playwright for efficient and future-proof web scraping in Python. TL;DR — Key Takeaways 2026 Use BeautifulSoup with "html.parser" for simple static pages Use parsel.Selector when you need both CSS and XPath Use Playwright for JavaScript-render...

Category: Web Scrapping • From: Setting up a Selector in Python 2026: Best Practices for Web Scraping

Q67. What are the best practices for Cost Optimization Techniques for Agentic AI Systems in 2026 in modern Python development?

Running Agentic AI systems can become extremely expensive in 2026. A single complex multi-agent workflow can easily consume thousands of tokens and cost several dollars per request. Without proper cost optimization strategies, production Agentic AI deployments 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 Agentic AI systems are naturally expensive because they typically involve: Multiple LLM calls per task Long context windows with memory and retrieved documents External tool usage and API calls Vector d...

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

Q68. 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

Q69. How does Removing missing values work? Give a practical example.

Removing missing values (using dropna() in pandas or drop_nulls() in Polars) is the quickest and cleanest way to prepare data for modeling — but it comes at a cost: every dropped row or column means permanently lost information. In 2026, you only drop when missingness is low, random, and non-critical. Otherwise, you risk introducing bias, reducing statistical power, or throwing away valuable signal. Here’s a complete, practical guide: when to drop (and when not to), how to do it safely and precisely, what to check before and after, and modern alternatives for speed on large data. 1. Basic & Targeted Dropping in Pandas import pandas as pd # Realistic example: customer data with common missing pat...

Category: Data Manipulation • From: Removing missing values

Q70. What are the best practices for Immutable or Mutable in modern Python development?

Immutable or mutable is a fundamental distinction in Python that determines whether an object's state (its value or contents) can be changed after creation. Immutable objects (int, float, bool, str, tuple, frozenset, bytes, range, etc.) cannot be modified — any "change" creates a new object and rebinds the name. Mutable objects (list, dict, set, bytearray, custom classes without __hash__ restriction, most user-defined objects) can be modified in-place, affecting all references to the same object. In 2026, understanding mutability is critical — it explains many subtle bugs (mutable defaults, shared state in pipelines, unexpected changes in data frames), impacts hashability (dict keys, set members), performance ...

Category: Writing Functions • From: Immutable or Mutable

Q71. How does LangGraph Human-in-the-Loop Patterns & Examples in 2026 (Approval, Interrupt, Resume + Guide) work? Give a practical example.

Updated March 16, 2026 : Covers LangGraph 0.3+ human-in-the-loop features (breakpoints, interrupt_before/after, Command(resume), editable state, approval nodes), examples with Llama-3.1-70B & Qwen-2.5-72B via vLLM, MotherDuck MCP tool integration, real-world latency & reliability notes, and 2026 best practices for production agents requiring human oversight. All code tested with uv + vLLM server, March 2026. LangGraph Human-in-the-Loop Patterns & Examples in 2026 (Approval, Interrupt, Resume + Guide) Even in 2026, the most reliable agentic systems still need humans in critical moments — approving high-stakes actions, correcting hallucinations, injecting domain knowledge, or overriding decisions. LangGraph makes...

Category: Data Sciences • From: LangGraph Human-in-the-Loop Patterns & Examples in 2026 (Approval, Interrupt, Resume + Guide)

Q72. What are the best practices for Cumulative statistics in modern Python development?

Cumulative statistics — running totals, max/min, products — are essential for understanding how data accumulates over time or rows. In Pandas, the family of .cumsum() , .cummax() , .cummin() , and .cumprod() methods make these calculations simple and efficient, turning raw sequences into powerful insights for tracking growth, balances, peaks, or multiplicative effects. In 2026, these methods are still core to time-series analysis, financial reporting, inventory tracking, and progressive metrics — especially when combined with groupby() for per-group running totals. 1. The Core Cumulative Methods .cumsum() — running sum (most common) .cummax() — running maximum (track all-time highs) .cummin(...

Category: Data Manipulation • From: Cumulative statistics

Q73. How does round() in Python 2026: Rounding Numbers + Modern Precision & Use Cases work? Give a practical example.

round() in Python 2026: Rounding Numbers + Modern Precision & Use Cases The built-in round(number, ndigits=None) function rounds a floating-point number to the specified number of decimal places (or to nearest integer if ndigits omitted). In 2026 it remains the standard, banker's-rounding tool for numeric formatting, reporting, ML metric display, financial approximations, and data cleaning — while exact decimal work still belongs to decimal.Decimal. Python 3.12–3.14+ improved float performance and precision handling, free-threading safety for math ops, and better integration with NumPy/JAX for array rounding. This March 24, 2026 update explains round() behavior, rounding modes, real-world patterns, common...

Category: Built in Function • From: round() in Python 2026: Rounding Numbers + Modern Precision & Use Cases

Q74. Explain 'List comprehension with range()' in detail. Why is it important in 2026?

List comprehension with range() is one of Python’s most common and powerful patterns — it lets you generate new lists of numbers (or derived values) in a single, concise line by combining range() (for number sequences) with transformations and optional filtering. It’s faster, cleaner, and more readable than a for loop + append() for simple cases, and it’s used everywhere from quick math lists to data generation, indexing, and preprocessing. In 2026, this pattern remains a Python staple — especially with type hints, large ranges in data science, and modern tools like Polars. Here’s a complete, practical guide to using list comprehensions with range() : basic generation, filtering, real-world patterns, ...

Category: Data Science Tool Box • From: List comprehension with range()

Q75. Explain 'hex() in Python 2026: Hexadecimal Representation + Modern Use Cases & Best Practices' in detail. Why is it important in 2026?

hex() in Python 2026: Hexadecimal Representation + Modern Use Cases & Best Practices The built-in hex() function converts an integer to a lowercase hexadecimal string prefixed with "0x". In 2026 it remains a simple yet essential tool for bit-level debugging, color codes, memory addresses, binary protocols, cryptography (byte → hex), low-level I/O, and educational purposes where human-readable hex output is needed. With Python 3.12–3.14+ delivering faster integer-to-string conversions, better free-threading support for concurrent formatting, and growing use in blockchain, hardware interfacing, and ML feature visualization, hex() is more relevant than ever. This March 23, 2026 update explains how hex() beha...

Category: Built in Function • From: hex() in Python 2026: Hexadecimal Representation + Modern Use Cases & Best Practices

Q76. Explain 'Python Datetime & Timezones 2026: zoneinfo vs Pendulum Tutorial + Best Practices' in detail. Why is it important in 2026?

Python Datetime & Timezones in 2026 — zoneinfo vs Pendulum: Full Tutorial & Best Practices Working with dates, times, and especially timezones in Python can be surprisingly painful — DST bugs, naive vs aware confusion, ambiguous times during fall-back, and inconsistent offsets across libraries. In 2026, with global apps, logging, scheduling, and data pipelines everywhere, getting this right is non-negotiable. I've dealt with timezone nightmares in production ETL jobs, user-facing dashboards, and earthquake timestamp analysis. After testing both native zoneinfo (Python 3.9+) and Pendulum extensively in 2025–2026, I now default to zoneinfo for most work — but reach for Pendulum when I need human-friendly ...

Category: Datatypes • From: Python Datetime & Timezones 2026: zoneinfo vs Pendulum Tutorial + Best Practices

Q77. Explain 'Slashes and Brackets in Web Scraping with Python 2026: XPath vs CSS Explained' in detail. Why is it important in 2026?

Slashes and Brackets in Web Scraping with Python 2026: XPath vs CSS Explained When learning web scraping, many beginners get confused by slashes (`/`, `//`) and brackets (`[]`, `()`) in selectors. These symbols are the core syntax of **XPath** and behave differently from CSS selectors. In 2026, understanding when to use slashes and brackets helps you write more powerful, precise, and maintainable scrapers. This March 24, 2026 guide clearly explains the meaning and usage of slashes and brackets in modern Python web scraping using both XPath and CSS. TL;DR — Key Takeaways 2026 / = direct child (like CSS >) // = descendant anywhere in the document (very powerful) [] = attribute or condition filt...

Category: Web Scrapping • From: Slashes and Brackets in Web Scraping with Python 2026: XPath vs CSS Explained

Q78. Explain 'Moving calculations above a loop' in detail. Why is it important in 2026?

Moving calculations above a loop is one of the simplest and most effective optimizations in Python — it eliminates redundant computations by performing expensive or repeated operations once outside the loop instead of recalculating them on every iteration. In 2026, this technique remains essential for improving runtime performance, reducing CPU usage, and making code cleaner — especially in data processing, numerical simulations, list building, filtering, or any loop-heavy code where the same value or transformation is reused across iterations. Here’s a complete, practical guide to moving calculations above loops: why it matters, basic patterns, real-world examples, performance gains, and modern best practic...

Category: Efficient Code • From: Moving calculations above a loop

Q79. What are the best practices for Chunking & filtering together in modern Python development?

Chunking & filtering together is one of the most powerful memory-efficient techniques for processing large CSV files in pandas — read the file in small chunks with chunksize , immediately filter each chunk to keep only relevant rows, perform transformations or aggregations on the filtered data, and either append results incrementally or write them to disk/database. This approach avoids loading the entire file into memory, drastically reduces peak RAM usage, speeds up processing by discarding unneeded data early, and enables scalable ETL, cleaning, subsetting, or feature engineering on gigabyte-scale files. In 2026, chunking + filtering is a must-know pattern — especially when combined with pd.concat , to_pa...

Category: Parallel Programming With Dask • From: Chunking & filtering together

Q80. What are the best practices for Using zip() in modern Python development?

Using zip() is one of Python's most elegant and frequently used tools for parallel iteration — it lets you loop over multiple iterables (lists, tuples, strings, ranges, etc.) at the same time, pairing corresponding elements together into tuples. It stops at the shortest iterable, making it perfect for combining related data, transposing matrices, merging columns, or processing aligned sequences without manual indexing. In 2026, zip() remains a core Pythonic pattern — used constantly with for loops, comprehensions, unpacking, and modern libraries like pandas/Polars. Here's a complete, practical guide to using zip() : basic parallel iteration, real-world patterns, handling unequal lengths, and best practi...

Category: Data Science Tool Box • From: Using zip()

Q81. What are the best practices for Counting Made Easy in Python: Harness the Power of Counting Techniques in modern Python development?

Counting Made Easy in Python: Harness the Power of Counting Techniques is one of the most frequent and powerful operations in data processing — from frequency analysis and histogram building to deduplication, validation, and feature engineering. Python offers elegant, efficient, and expressive ways to count elements, occurrences, unique values, conditional matches, and more. In 2026, counting has evolved dramatically: Polars dominates for ultra-fast columnar counts, collections.Counter remains the gold standard for frequency maps, and NumPy / Dask handle massive datasets with vectorized or distributed operations. This guide covers every practical technique — from basics to high-performance patterns — wit...

Category: Datatypes • From: Counting Made Easy in Python: Harness the Power of Counting Techniques

Q82. What are the best practices for Calculating summary stats across columns in modern Python development?

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

Q83. What are the best practices for A decorator factory in modern Python development?

A decorator factory in Python is a higher-order function that takes parameters and returns a decorator — enabling customizable, parameterized decorators like @timer(message="Exec time") , @retry(attempts=3) , or @repeat(5) . The factory accepts arguments at decoration time, then returns the actual decorator that wraps the target function. This pattern separates configuration from wrapping logic, making decorators flexible, reusable, and declarative. In 2026, decorator factories are everywhere — powering configurable logging, timing with custom messages, retry/backoff policies, caching with maxsize, validation rules, rate limiting, and more in FastAPI, Flask, Celery, pandas/Polars pipelines, ML training loop...

Category: Writing Functions • From: A decorator factory

Q84. What are the best practices for Named placeholders in modern Python development?

Named placeholders make string formatting in Python more readable, maintainable, and self-documenting — instead of relying on positional indices {0} , {1} (which can become confusing with many variables or reordering), you use meaningful names like {name} , {age} directly in the format string and pass values via keyword arguments to .format() or f-strings. In 2026, named placeholders are especially valuable in dynamic templates (from config files, databases, or user input), internationalized strings, logging formats, report generation, and codebases with many variables where clarity trumps brevity. While f-strings dominate for static formatting, named placeholders shine when the format string is genera...

Category: Regular Expressions • From: Named placeholders

Q85. Explain 'Replacing missing values' in detail. Why is it important in 2026?

Replacing missing values (imputation via fillna() in pandas or fill_null() in Polars) is often the best compromise when dropping rows/columns would destroy too much data. The key is choosing the right strategy: simple fills preserve row count but can distort distributions; advanced methods preserve realism but take more compute. In 2026, start simple (mean/median/mode/ffill), then move to model-based imputation (KNN/MICE) when accuracy matters. Always visualize before and after — never assume the fill is harmless. 1. Basic Replacement in Pandas import pandas as pd import seaborn as sns import matplotlib.pyplot as plt # Realistic example: sales data with common missing patterns data = { '...

Category: Data Manipulation • From: Replacing missing values

Q86. How does Working with Datetime Components and Current Time in Python work? Give a practical example.

Working with Datetime Components and Current Time in Python is a core skill for any developer handling timestamps, scheduling, logging, data analysis, or time-based features. The datetime module (and its companions date , time , timedelta , timezone) provides comprehensive, timezone-aware tools for date/time manipulation. In 2026, Python's datetime ecosystem is mature: zoneinfo (stdlib since 3.9) replaces pytz for modern timezone support, Polars/Dask offer fast columnar datetime operations, and Pydantic/dataclasses integrate datetime validation seamlessly. This guide covers component extraction, current time retrieval, formatting/parsing, timezone handling, real-world patterns (earthquake timestamp ...

Category: Datatypes • From: Working with Datetime Components and Current Time in Python

Q87. How does Regex metacharacters work? Give a practical example.

Regex metacharacters are the special symbols in Python's regular expressions (via the re module) that have predefined meanings beyond their literal character — they define patterns, repetition, positions, groups, character classes, and more. These metacharacters make regex powerful for matching complex text structures like emails, dates, phone numbers, log formats, HTML tags, or custom delimiters. In 2026, understanding metacharacters remains fundamental — they are used constantly in data validation, text extraction, cleaning, parsing, and vectorized pandas/Polars string column operations. Knowing when and how to escape them (with \ ) prevents syntax errors and unintended matches. Here's a complete, pract...

Category: Regular Expressions • From: Regex metacharacters

Q88. 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

Q89. What are the best practices for any() in Python 2026: Check If Any Element Is True + Modern Patterns & Use Cases in modern Python development?

any() in Python 2026: Check If Any Element Is True + Modern Patterns & Use Cases The built-in any() function returns True if at least one element in an iterable is truthy (and False if the iterable is empty). It’s the logical counterpart to all() and remains one of the most elegant, short-circuiting, and performant tools for existence checks, early-exit conditions, and validation in Python code. In 2026, with type hints, FastAPI route guards, data anomaly detection, ML early stopping, and async workflows, any() is more valuable than ever — especially in generator expressions and lazy evaluation. This March 23, 2026 update explains how any() behaves today, real-world patterns, performance notes, an...

Category: Built in Function • From: any() in Python 2026: Check If Any Element Is True + Modern Patterns & Use Cases

Q90. How does Built-in function: map() with Efficient Code work? Give a practical example.

Built-in function: enumerate() with Efficient Code is one of Python’s best tools for clean, readable, and performant iteration when you need both the item and its position (index). Instead of manually using range(len(iterable)) and indexing (which is error-prone, slower, and less Pythonic), enumerate() pairs each element with its index in one step — faster, safer, and more readable. In 2026, enumerate() remains a core pattern — used everywhere from simple lists to pandas DataFrames, file line numbering, and parallel processing with zip() . Here’s a complete, practical guide to using enumerate() efficiently: why it beats manual indexing, real-world patterns, performance advantages, and modern best pr...

Category: Efficient Code • From: Built-in function: map() with Efficient Code

Q91. What are the best practices for Building Your First Scrapy Spider in 2026 – Modern Python Web Scrapping Guide in modern Python development?

Scrapy remains the most powerful open-source framework for structured web scrapping in Python in 2026. This updated guide shows how to create a clean, modern "spider" (crawler) using Scrapy 2.14+, Python 3.11–3.13, and current best practices including async support. What is a Scrapy Spider? A spider defines how to crawl a site (start URLs), how to parse pages, and what data to extract. In 2026, spiders are often written with async methods for better performance. Step 1 – Project Setup (2026 style) pip install scrapy scrapy startproject classy_spider_2026 cd classy_spider_2026 Step 2 – Create the Spider (modern async style) # classy_spider_2026/spiders/quotes.py import scrapy class Quote...

Category: Web Scrapping • From: Building Your First Scrapy Spider in 2026 – Modern Python Web Scrapping Guide

Q92. Explain 'Functional Approaches using Dask Bags' in detail. Why is it important in 2026?

Functional Approaches Using dask.bag bring the elegance of functional programming to parallel processing of large, unstructured, or semi-structured datasets — leveraging pure functions (map , filter , reduce / fold) that are composable, stateless, and naturally parallelizable across chunks, cores, or clusters.

In 2026, Dask Bags remain the ideal tool for line-based or record-based data (logs, JSONL, text corpora, sensor streams, earthquake metadata) where tabular structure is absent or emerges late — enabling clean, declarative pipelines that scale effortlessly while integrating with Dask DataFrames (for tabular output) and xarray (for labeled arrays). Here's a complete, practical guide to functional app...

Category: Parallel Programming With Dask • From: Functional Approaches using Dask Bags

Q93. How does Using decorator @-notation work? Give a practical example.

Using decorator @-notation is Python's cleanest, most idiomatic way to apply decorators — the @ symbol placed directly above a function (or class) definition replaces the traditional manual wrapping: func = decorator(func) . It's syntactic sugar that makes code more readable, declarative, and self-documenting, especially when stacking multiple decorators or using parameterized ones. In 2026, @-notation is universal — every major library (FastAPI, Flask, pytest, Typer, Pydantic, Celery, pandas/Polars extensions) uses it for logging, timing, caching, auth, validation, retry, dependency injection, and more. Mastering @-notation (and its nuances with arguments) lets you write concise, professional, production-r...

Category: Parallel Programming With Dask • From: Using decorator @-notation

Q94. How does Different statistics in a pivot table work? Give a practical example.

You can calculate different statistics in a pivot table by passing a different aggregation function to the aggfunc parameter in the pivot_table() method. Here's an example: import pandas as pd # create a pandas DataFrame df = pd . DataFrame ({ 'Region' : ['North' , 'North' , 'South' , 'South' , 'West' , 'West'], 'Salesperson' : ['Alice' , 'Bob' , 'Charlie' , 'Dave' , 'Eve' , 'Frank'], 'Sales' : [100 , 200 , 150 , 50 , 75 , 125], 'Profit' : [20 , 50 , 30 , 10 , 15 , 25]}) # create a pivot table that summarizes the sales and profit data by region and salesperson ...

Category: Data Manipulation • From: Different statistics in a pivot table

Q95. How does Timezone Hopping with Pendulum: Seamlessly Manage Time across Different Timezones work? Give a practical example.

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

Q96. How does Dict comprehensions work? Give a practical example.

Dict comprehensions are Python’s concise, expressive way to create new dictionaries from iterables — similar to list comprehensions but producing key-value pairs instead of single values. The syntax {key_expr: value_expr for item in iterable if condition} lets you transform, filter, and build dicts in a single readable line. In 2026, dict comprehensions are essential — used constantly for data restructuring, filtering JSON/API results, creating lookup tables, inverting mappings, and preparing data for pandas/Polars or databases. Here’s a complete, practical guide to dict comprehensions: basic creation, filtering, real-world patterns, conditional values, and modern best practices with type hints, readabilit...

Category: Data Science Tool Box • From: Dict comprehensions

Q97. How does Allocating memory for an array work? Give a practical example.

Allocating memory for an array in Python is rarely done explicitly like in lower-level languages — Python handles memory management automatically via reference counting and garbage collection. However, when you need to create fixed-size, efficient, or typed arrays (especially for performance or interoperability), you have several options: built-in list (dynamic, most common), array.array (compact, typed), NumPy arrays (high-performance, vectorized), or Polars Series/DataFrames (modern columnar data). In 2026, NumPy and Polars dominate for numerical/scientific work due to speed, memory efficiency, and broadcasting, while plain lists remain the go-to for general-purpose dynamic arrays. Understanding allocati...

Category: Parallel Programming With Dask • From: Allocating memory for an array

Q98. What are the best practices for Creating Sets in Python: Harnessing the Power of Unique Collections in modern Python development?

Creating Sets in Python: Harnessing the Power of Unique Collections introduces one of Python’s most efficient and mathematically elegant data structures — the set. Sets store unique, hashable elements with no duplicates and no inherent order (though insertion order is preserved in CPython since 3.7). They excel at membership testing (O(1) average time), deduplication, and set algebra (union, intersection, difference, symmetric difference) — making them ideal for uniqueness enforcement, overlap detection, filtering invalid data, and comparing collections. In 2026, sets remain indispensable in data science (unique values, categorical filtering), software engineering (permission sets, tag collections, cache keys)...

Category: Datatypes • From: Creating Sets in Python: Harnessing the Power of Unique Collections

Q99. Explain 'Using nonlocal' in detail. Why is it important in 2026?

Using nonlocal is how Python lets a nested (inner) function modify variables from its enclosing (outer) function’s scope. Without nonlocal , assigning to an outer variable inside the inner function creates a new local variable instead — leading to the infamous UnboundLocalError or silent bugs. In 2026, nonlocal remains essential for closures, counters, factories, decorators, and any time you need persistent state without classes or globals. Here’s a complete, practical guide to using nonlocal : how it works, when to use it, real patterns, pitfalls, and modern best practices with type hints and clarity. Start with the basics. In this classic counter factory, the inner function needs to modify the oute...

Category: Data Science Tool Box • From: Using nonlocal

Q100. How does Iterating with file connections work? Give a practical example.

Iterating with file connections is one of the cleanest and most efficient ways to read files in Python. When you open a file with open() , the file object itself is an iterable — a for loop steps through it line by line automatically, without loading the entire file into memory. This pattern is perfect for processing large log files, CSVs, text data, or any line-based content — it’s fast, memory-safe, and very Pythonic. In 2026, iterating over files remains a core skill — especially with context managers (with), error handling, and modern tools like Polars for structured data. Here’s a complete, practical guide to iterating over files: basic line-by-line loops, stripping newlines, real-world patterns, a...

Category: Data Science Tool Box • From: Iterating with file connections