Faysal Aziz

Advanced CSE with Python

The Python Engineer's Toolbox: Advanced Skills for Modern Challenges in 2 days

FA-0034 · 2 days

Software Development

Introduction

In the rapidly advancing field of software engineering, proficiency in Python is more than a skill—it's an essential tool.

This two-day course on Advanced Computer Software Engineering with Python is designed to push the boundaries of what you can achieve with Python, targeting complex software solutions and performance optimizations. Python, known for its readability and efficiency, is a prime choice for tackling advanced concepts in software development, from intricate algorithmic designs to high-level asynchronous programming.

As industries lean towards automation, data analysis, and scalable web services, advanced Python skills are increasingly sought after. This course aims to provide seasoned developers with the knowledge to implement sophisticated programming techniques, optimize existing systems, and innovate with Python at the forefront of technology.

Learning outcomes

Participants will:

Master Advanced Python Features

1. **Comprehensive Understanding of Advanced Features**: Participants will gain an in-depth knowledge of Python's advanced syntax and features, such as decorators, generators, context managers, and metaclasses.
2. **Practical Application in Real-World Scenarios**: Learn how to apply these advanced features in various software development contexts, enhancing the functionality and efficiency of applications.

Apply Design Patterns and Architectural Principles

1. **Incorporation of Design Patterns**: Gain the ability to identify and implement appropriate design patterns that solve specific design problems and improve the maintainability of the software.
2. **Architectural Principles for Scalable Applications**: Understand how to architect applications using principles like modularity, separation of concerns, and encapsulation to build systems that are scalable, robust, and easy to update.

Utilize Advanced Concurrency Models

1. **Mastering Multithreading and Multiprocessing**: Acquire skills in effectively using Python's threading and multiprocessing modules to perform concurrent execution of code.
2. **Error-Free and Efficient Code**: Learn techniques to ensure code reliability and safety when dealing with concurrent executions, avoiding common pitfalls like race conditions and deadlocks.

Enhance Coding Toolkit with Python's Libraries and Frameworks

1. **Deep Dive into Standard and Third-Party Libraries**: Explore a range of Python libraries such as NumPy for numerical computations, pandas for data manipulation, and Matplotlib for data visualization.

Develop a Systematic Approach to Solving Complex Problems

1. **Algorithmic Thinking and Optimization**: Enhance problem-solving skills with a focus on algorithm design and optimization, learning how to approach problems logically and efficiently.
2. **Code Reviews and Collaborative Problem Solving**: Foster an environment of peer learning through code reviews, where participants critique and improve each other's code, leading to higher code quality and better team collaboration.

Additional Outcomes

1. **Integration of Advanced Data Structures**: Learn about advanced data structures and how to implement them in Python for more efficient data handling and manipulation.
2. **Performance Tuning and Profiling**: Acquire the ability to profile Python applications to identify bottlenecks and apply optimization techniques to enhance performance.
3. **Security Best Practices**: Understand security best practices in Python programming, learning how to secure applications against common vulnerabilities like SQL injection and cross-site scripting (XSS).

Prerequisites

This course requires participants to have:

- Profound knowledge of Python, including its standard library.
- Practical experience with object-oriented programming and Python's approach to OOP.
- Prior exposure to web development frameworks in Python and RESTful API integration.
- Comfort with advanced data manipulation using libraries such as pandas and NumPy.
- Familiarity with software design patterns and architectural principles as applied in Python.
- Experience with Python's concurrency tools, including threading and multiprocessing modules.
- Understanding of asynchronous programming in Python using asyncio.

- Capability to utilize advanced Python features such as decorators, generators, and context managers.
- Previous involvement in a software development project where Python was the primary programming language.
- Knowledge of unit testing and debugging in Python, including experience with frameworks such as PyTest or unittest.
- Experience with version control systems, preferably Git, in the context of Python projects.
- Understanding of performance optimization and proficiency in using profiling tools in Python.
- Knowledge of security practices relevant to Python and web applications.
- Previous use of integrated development environments (IDEs) and code editors optimized for Python development, such as PyCharm, VS Code, or Jupyter / Colab Notebook.
- Experience using Unix like operating systems and good understanding of overall OS standards.

Training outline

1. Advanced Python Techniques

Dynamic Typing and Memory Management

1. Deep dive into Python's memory management and garbage collection
2. Best practices for dynamic typing and avoiding common pitfalls

Functional Programming Tools

1. In-depth usage of lambda, map, filter, and reduce
2. Techniques for integrating functional programming paradigms into Python code

2. Software Design Patterns

Creational Patterns

Factory Method

1. **Dynamic Object Creation**: Explore the concept of the Factory Method, which allows a class to delegate the responsibility of instantiating objects to subclasses. This pattern is particularly useful when there is a need to create objects whose specific types cannot be anticipated by the code that needs to instantiate them.
2. **Role of Factories in Abstracting Instantiation Logic**: Understand how factory methods can encapsulate the creation logic of objects, leading to code that is more modular, adaptable, and easy to maintain. Examples will demonstrate creating diverse objects adhering to a common interface or base class.

Abstract Factory

1. **Systematizing the Use of Factory Methods**: Learn about the Abstract Factory pattern, which provides an interface for creating families of related or dependent objects without specifying their concrete classes. This pattern is ideal for scenarios where systems need to be

independent from how the products are created, composed, and represented.

2. **Application in Modular Design**: Case studies showing how abstract factories can manage sets of factory methods, each of which is designed to create different kinds of objects based on the context of the application.

Structural Patterns

Composite

1. **Managing Tree-Like Object Structures**: This pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Dive into how composites let clients treat individual objects and compositions of objects uniformly.
2. **Unified Treatment Application**: Detailed examples will show how the Composite pattern can simplify complex tree structures, making it easier to add new types of components while maintaining a clean codebase.

Proxy

1. **Controlling Object Access and Optimization**: A Proxy pattern provides a surrogate or placeholder for another object to control access to it. This is useful for managing operations that are resource-intensive or need to be secured.
2. **Practical Use Cases**: Learn through examples how proxies can handle lazy initialization, access control, logging, and smart reference counting, which are essential for optimizing interactions and resource management.

Behavioral Patterns

Mediator

1. **Reducing Direct Communications Between Objects**: This pattern introduces a single component, the mediator, which takes over control of method calls between objects. By doing so, it removes the need for components to reference each other explicitly, which decreases coupling and adheres to the principle of least knowledge.
2. **Control Interactions**: Explore scenarios where mediators can be used to streamline communications in complex systems, such as in GUI development or when implementing system-wide actions that involve various object interactions.

Visitor

1. **Adding New Operations to Objects Without Altering Their Classes**: The Visitor pattern allows you to add new operations to existing object structures without modifying the structures themselves. This is particularly useful in applications where functionalities need to be extended frequently.
2. **Separation of Concerns**: Examples will demonstrate how visitors can separate algorithms from the elements on which they operate, thus making additions of new operations easy and keeping the existing classes unchanged.

3. Concurrency and Parallelism

Advanced Threading Techniques

Thread Pools and Managing Thread Life Cycles

1. **Using ThreadPoolExecutor**: Learn to manage a pool of threads efficiently using the concurrent.futures.ThreadPoolExecutor, which handles the allocation and recycling of threads for you. Understand how to submit tasks to the pool and collect results asynchronously, optimizing the use of system resources and improving application throughput.
2. **Lifecycle Management**: Explore best practices for starting, pausing, resuming, and stopping threads safely. Discuss techniques to detect and handle thread states to prevent deadlocks and ensure smooth execution.

Thread Safety and Synchronization Tools in Python

1. **Locks, Semaphores, and Conditions**: Deep dive into Python's threading module, focusing on synchronization primitives like Lock, RLock, Semaphore, and Condition objects. Learn how these tools can be used to prevent race conditions and ensure threads do not corrupt shared data.
2. **Deadlock Prevention**: Strategies to avoid deadlocks, such as ordering locks and using a with statement to ensure that mutexes are always released.

Multiprocessing and Parallel Execution

Techniques for Dividing Tasks Across Processors

1. **Using Multiprocessing Pool**: Similar to thread pools but for processes, using the multiprocessing.Pool to manage separate Python interpreter processes, thereby bypassing the GIL and utilizing multiple CPU cores.
2. **Task Decomposition**: Best practices for decomposing tasks to be executed in parallel, considering data dependencies and minimizing the need for inter-process communication.

Understanding the Global Interpreter Lock (GIL) and Its Implications on Multi-Threading

1. **Role of the GIL**: Explain why Python uses the GIL, which is to avoid conflicts between threads in memory management (specifically, reference counting) in CPython.
2. **Implications for Multi-Threading**: Discuss how the GIL can lead to performance bottlenecks in CPU-bound multi-threaded programs and explore strategies to mitigate this, such as using the multiprocessing module to create processes instead of threads for concurrent execution.

Asynchronous Programming

Event Loop Management with Asyncio

1. **Understanding the Event Loop**: Learn the core concept of the asyncio library's event loop, which is central to asynchronous programming in Python. Explore how the event loop executes tasks, handles I/O events, and manages asynchronous routines.
2. **Handling Coroutines and Tasks**: Deep dive into creating and managing coroutines with async def, await, asyncio.create_task(), and asyncio.gather() for concurrent execution of tasks.

Real-World Applications of Async/Await Patterns

1. **Web Scraping and Network Communication**: Practical applications of asyncio for asynchronous web scraping and handling multiple network connections concurrently.
2. **Integrating Async/Await in Web Servers**: Implementing asynchronous request handling in web frameworks like Sanic and AIOHTTP which are built on top of asyncio, demonstrating how to build high-performance Python web servers.

4. Recursion and Algorithms

Mastering Recursion

Understanding Recursion in Python

1. **Concepts and Execution**: Delve into the fundamentals of recursion, how Python handles recursive calls, and the underlying stack-based execution model. This includes understanding how Python allocates stack frames for each recursive call and the limitations imposed by maximum recursion depth.
2. **Stack Memory Implications**: Explore the implications of recursive calls on stack memory, including potential stack overflow errors, and how to mitigate these risks using tail recursion optimization techniques in Python.

Practical Applications and Writing Efficient Recursive Functions

1. **Designing Recursive Algorithms**: Learn to convert complex problems into recursive solutions by breaking them down into base cases and recursive steps.
2. **Efficiency Best Practices**: Enhance the efficiency of recursive functions using memoization techniques to avoid redundant calculations, effectively reducing the computational complexity.

Algorithm Optimization

Space-Time Trade-offs in Algorithms

1. **Understanding Trade-offs**: Discuss how algorithm designs often trade computational time for space (memory usage) or vice versa. Analyze how different approaches can be used depending on the constraints of the problem and the resources available.
2. **Memoization**: Deep dive into memoization as a specific technique to improve the runtime of algorithms by storing results of expensive function calls and reusing them when the same inputs occur again. This is particularly useful in recursive algorithms to prevent the exponential time complexity often associated with multiple recalculations of the same data.

Advanced Sorting and Searching Algorithms Tailored for Python's Strengths

1. **Sorting Algorithms**: Explore advanced sorting techniques beyond Python's built-in sort methods, such as Tim Sort (Python's default), merge sort, quicksort, and heap sort. Understand when and why to use each sorting algorithm based on their performance characteristics and Python's optimization.
2. **Searching Algorithms**: Master advanced searching algorithms, including binary search on sorted data, depth-first search (DFS) and breadth-first search (BFS) for tree and graph traversals. Learn how these algorithms can be optimized in Python using data structures like lists, dictionaries, and sets for efficient lookup and retrieval.

5. Advanced Libraries and Frameworks

1. **Handling Large Data Sets**: Learn to manage and process large arrays and matrices with NumPy efficiently, including techniques for memory management, vectorization for speed, and applying ufuncs (universal functions).

2. **Advanced Indexing and Slicing**: Master complex indexing techniques which allow for effective data selection and manipulation, essential for data analysis and preprocessing tasks.
3. **Numerical Operations and Optimizations**: Explore numerical optimization techniques, including solving linear algebra problems and integrating C/C++ or Fortran code for performance critical operations.

Utilizing SciPy for Scientific and Technical Computing

1. **Module-Specific Deep Dive**: Detailed exploration of SciPy modules such as optimize for optimization algorithms, stats for statistical functions, and sparse for sparse matrix handling.
2. **Integration and Differential Equations**: Learn how to solve ordinary differential equations (ODEs) using SciPy's integrative functions to model real-world phenomena in engineering and science.
3. **Signal Processing**: Techniques for signal filtering, convolution, and spectral analysis, which are crucial for data analysis in fields such as neuroscience or engineering.
4. **Image Processing with SciPy and NumPy**: Apply advanced image processing techniques, including edge detection, image filtering, and morphological transformations to handle image data effectively.

6. Best Practices and Real-World Application

Code Quality and Maintainability

1. Strategies for writing clean, understandable, and maintainable code
2. Advanced use of linters and formatters in a professional Python environment

Problem Solving and Case Studies

1. Analyzing real-world problems and modeling solutions using advanced Python
2. Peer reviews and collaborative exercises to refine solutions and approaches

This meticulously crafted outline ensures that participants gain a deep understanding of Python's advanced capabilities and are well-prepared to apply these skills in cutting-edge software development projects.

© 2026 Faysal Aziz · Kuala Lumpur