Advanced Python Programming

24 Hours

Course Overview:

Python has seen incredible growth as a preferred language in many diverse fields such as DevOps, Data analysis, Machine learning and AI, Mapping and GIS, testing and automation, and web applications. Its incredible flexibility and amazing libraries has made a mark on the development landscape.

The difference in productivity from knowing basic python and mastering advanced techniques can be staggering. Investing the time to learn these techniques can be a real edge in your skillset.

This workshop is designed to build deep understanding in a wide array of advanced techniques and applications. The workshop uses extensive in-class practice to retain and refine this understanding.

Course Objectives:

- To gain an in-depth under-the-hood understanding of python OO and functional programming
- Master advanced techniques such as closures, decorators, protocols, abstract base classes, introspection and multiple inheritance
- Writing shorter, more expressive and more beautiful code
- Have the toolset needed to develop your own incredible containers, frameworks and modules from the ground-up
- Practice wide set of incredibly useful applications skills, such as DB and SQL with SQLAlchemy, test driven development with pytest and unittest.mocks and dependency management using pyenv

Who Should Attend:

This course is intended for developers with intermediate or advanced python knowledge who want to take their skills to the next level.

Required Skills:

- Python essentials: functions, comprehensions, classes, OOP
- Familiarity with at least one python IDE such as VS Code, PyCharm, Jupyter notebook

Course Contents:

Part I - Deep functional and OO programming in python

- Python programming quick recap
  - List comprehensions & friends
  - Variadic functions - *args, **kwargs
  - Lambda
  - Basic classes
- Closures and decorators
  - Higher order functions
  - Closures
    - Closures as replacement for functor objects
    - Example: Event handling
  - Decorators
    - Simple decorators

- Variadic decorators
- Standard decorators: @functools.wraps, @classmethod, @staticmethod, and @property
- Examples: Profiling decorator, Debugging decorator, Plugins
- Decorating classes. Example: @dataclass
- Nesting Decorators
- Decorators With Arguments

- Polymorphism, protocols, abcs
  - Polymorphism
    - Classic OOP Polymorphism
    - Duck typing
  - Protocols
  - Abstract base classes (ABCs)
    - Enforcing a protocol
    - Checking for a protocol
    - Registering types
- Writing our own container types
  - Iterator protocol and iterable
    - for loops and iterables
    - Iterating with iter() and sentinel values
    - Example: VersionedObject
  - Sequence protocol
    - Inheriting from collections.abc.Sequence
    - Example: MyRange
  - Mapping protocol
    - Example: ObservableMapping
- Python OOP in-depth
  - Constructors are inheritable
  - Bound and unbound methods
  - Monkey-patching methods
  - Where are all the attributes coming from?
    - __dict__, __slots__
    - __class__, __base__, __bases__
    - __getattr__, __getattributes__
  - Testing if an object has an attribute: getattr()
  - inspecting objects: dir(), inspect
  - Method Resolution Order: mro()
  - Accessing your base class: direct vs super()
  - class scope: code and variables
  - using __getattr__() to dynamically support more attributes
- Multiple inheritance and super
  - The need for super()
  - Challenges - “super considered harmful”
  - Solutions - “super considered super”
  - Example: superobject

- Its fun to be eval
  - Running code from strings: eval, exec, compile
  - Example: parse JSON using eval
  - Controlling the environment: locals, globals
  - Example: rule engine
  - Caveat: security risk
- Complex and nested list comprehensions
  - One simple rule
  - Flattening a nested list
  - Transforming a nested list
  - Making a nested list out of a flattened one
- Classic design patterns in python
  - Prototype
  - Singleton
  - Proxy
  - Composition
  - Abstract Factory
  - Resource Acquisition Is Initialization (RAII): @contextlib.contextmanager

Part II - Advanced applications

- Intro to test driven development with pytest and unittest.mock
  - Pytest
    - Writing test_ functions
    - Asserts
    - Print redirection
    - Dependency injection with fixtures
    - Multiple params
  - Mocking
    - Mock asserts
    - create_autospec
    - Mock patching
- Intro to package management with pipenv
  - Standard library
  - Site-packages: global / per-user
  - Helper modules: inspect, site
  - Pipenv
    - Intro to PEP 508 constraints
    - Working with multiple projects, where each has their own decencies
    - Having reproducible deterministic development / production environments
    - Updating dependencies to fix security issues
    - Development-only packages
    - Tying it all together
- Intro to database access
  - SQL recap: SELECT, FROM, WHERE, JOIN, GROUP BY, ORDER BY

- DB-API
- Sqlite3
- Pyodbc
- SQLAlchemy
    - Mapping classes to tables with declarative classes
    - Querying with ORM session
    - Reflecting an existing database with automap and sqlacodegen