Best Practices for Structuring a Python CLI Application

4 min read · Aug 25, 2023

Ernest Winata Follow

This blog post outlines some best practices for structuring a Python CLI application.

CLI (Command-Line Interface), is a program that allows users to interact with a computer's operating system by entering commands as text. These commands are typically executed in a terminal or command prompt. CLI applications offer a way to perform various tasks without the need for a graphical user interface (GUI).

The CLI application that I built for this project offers a user-friendly way to access and manipulate data related to countries and their iconic landmarks. By following a structured approach, I have designed the application to not only perform its intended function but also serve as a testament to the importance of proper project organization and the use of essential Python libraries.

When building the codes, I kept in mind the principle of "separation of concerns," which is the practice of breaking down a project into distinct, manageable

- Readability: a well-organized code is easier to read, understand, and navigate.
- Maintainability: when code is logically organized, making updates and fixing bugs becomes easier.
- Scalability: making it easier for engineers to implement future changes and expansions.
- Collaboration: separating concerns makes it easier for co-workers to work on different parts of the project.

In this project, I used a structured approach by dividing the code into distinct modules that handle different aspects of the application:

1. Models Module (models.py): to manage the database and to define data schema, I have created this module to define SQLAlchemy models for ‘Country’ and ‘Landmark’.
2. CLI Commands Module (cli.py): for handling user interactions and CLI commands, I have created a module named cli.py. This is where I used the Click library to define the commands users can use to interact with the application."

To help design user-friendly and intuitive commands, I used the Click library; Click provides decorators and tools for defining and handling commands. In my project, I used Click to define commands that allow users to interact with the database of countries and landmarks. Let’s take a look at how we defined the `add_country` command using Click:

```
@cli.command()
def add_country(country_name):
    session = Session()
    country = Country(name=country_name)
    session.add(country)
    session.commit()
    click.echo(f"Added {country_name} to the database.")
```

In this example, the `@cli.command()` decorator marks the `add_country` function as a Click command. The `click.argument()` decorator defines a command-line argument (`country_name`) that the user will provide when invoking the command. Inside the function, I have a new `Country` object using SQLAlchemy ORM, add it to the session, and commit the changes to the database. Finally, I provide user feedback using Click's `click.echo()` .

When users run the CLI script, Click interprets the decorators and function signatures to generate the appropriate command-line interface. This means that the function arguments, such as `country_name` in our example, are seamlessly translated into command-line arguments. Furthermore, users are able to receive immediate feedback in the event of errors.

Another Python library that I used for this project is SQLAlchemy. When working with databases, Object-Relational Mapping (ORM) tools like SQLAlchemy provides a powerful toolkit for interacting with databases. It allows us to define our database schema using Python classes, bringing together the world of object-oriented programming and relational databases seamlessly.

In my CLI application, I used SQLAlchemy to store information about countries and landmarks. I leveraged SQLAlchemy's declarative syntax to define two classes: `Country` and `Landmark` .

```
class Country(Base):
    __tablename__ = 'countries'

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)

class Landmark(Base):
    __tablename__ = 'landmarks'

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    city = Column(String)
    visited = Column(Boolean, default=False)

    country_id = Column(Integer, ForeignKey('countries.id'))
    country = relationship('Country', backref='landmarks')
```

These model classes serve as blueprints for the tables in the database. I defined the fields (columns) that each table should have and the relationships between them. For example, 'Landmark' class has a `country_id` column, establishing a relationship with the 'Country' class.

ORM tools like SQLAlchemy keeps database-related code isolated in its own module. This separation of concerns enhances the readability and maintainability of the codes.

In order to manage dependencies in one place, I used Pipenv to create a virtual environment. Issues can arise if different projects require different versions of the same library, or if the installed packages conflict with one another. Pipenv helps manage the project's stability and reproducibility. Each project gets its own isolated virtual environment, ensuring that dependencies don't interfere with each other. Additionally, by having Pipfile.lock, I would get the exact same versions of packages each time, preventing conflicts caused by version updates.

This blog post has briefly touched some of the best practices when it comes to structuring codes when building a CLI application. They not only increase the maintainability and extendibility of the project, but

also contribute to creating cleaner, more readable

E

Follow

Written by Ernest Winata

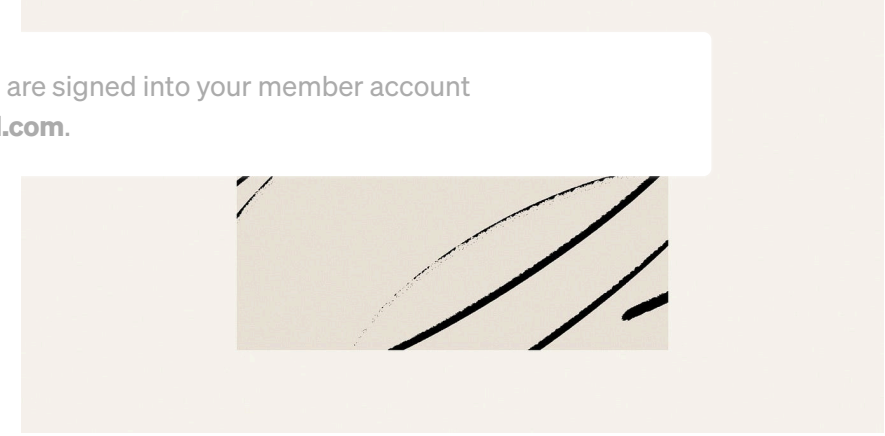
2 followers · 1 following

No responses yet

Twillbe Better

What are your thoughts?

More from Ernest Winata



Ernest Winata

Expanding your skills and horizons as an L1 at Amazon

I started my Amazon journey when I was first laid off in March 2020. I applied to the local Amazon Fulfillment Center (FC)...

May 3, 2023 1

Ernest Winata

Using Association Proxy as a Placeholder

For this blogpost, I will briefly outline the use of association proxy in SQLAlchemy. One of the stretch goals of the...

Oct 5, 2023

Ernest Winata

Building a Form in my React Project

In this blog post, I'm going to briefly touch on building a form in my React project. Forms are a crucial part of web...

Jul 1, 2023

Ernest Winata

Using .slice() to extract specific objects from a public API

For my Phase 1 project, I used The Dog API, containing a fun and informative huge list of breeds from all sorts of groups....

May 9, 2023

Recommended from Medium

Sohail Saifi

Why Japanese Developers Write Code Completely Differently (And Why It Works...

I've been studying Japanese software development practices for the past three years, and what I discovered completely...

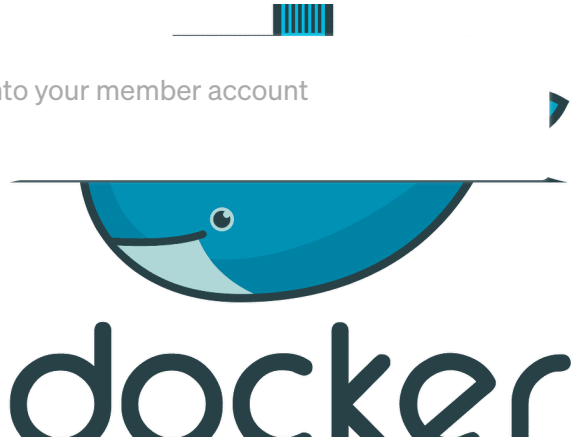
Jul 17 8.8K 236

Home
Following
Library
Stories

Recent

The Medium Blog
Anthony Alcaraz

Discover more writers and publications to follow.
See suggestions

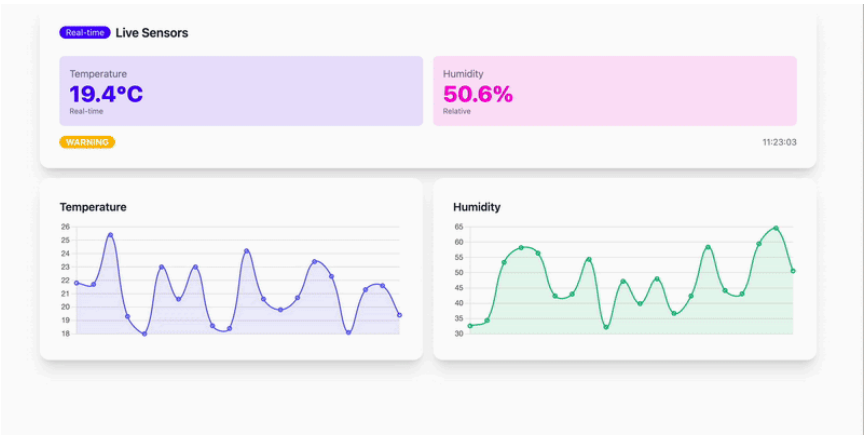


Abhinav

Docker Is Dead—And It's About Time

Docker changed the game when it launched in 2013, making containers accessible and turning "Dockerize it" into a...

Jun 9 3.3K 72

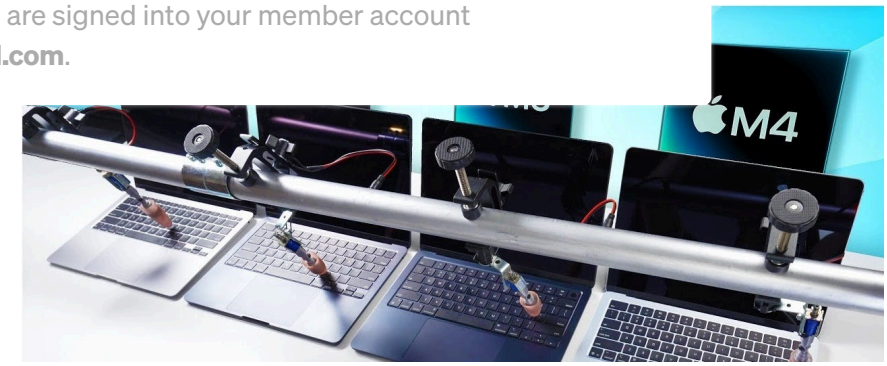


Daniel Boadzie

Building Real-Time Dashboards with FastAPI and HTMX

Why This Matters (And Why You Should Care)

Jun 30 422 5

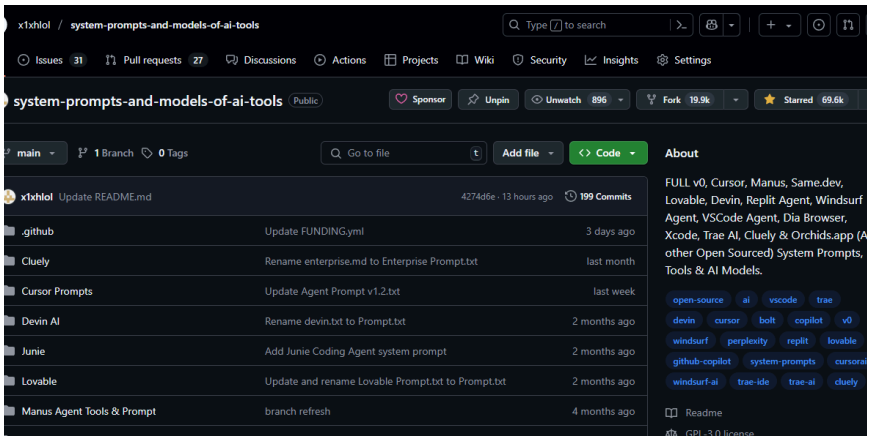


Faizan Saghir

M4 vs M3 vs M2 vs M1 MacBook Air For Coding: A Developer's Choice in 2025

Real Benchmarks, Dev Workloads, and the MacBook Air That Actually Delivers

Jun 1 101 5

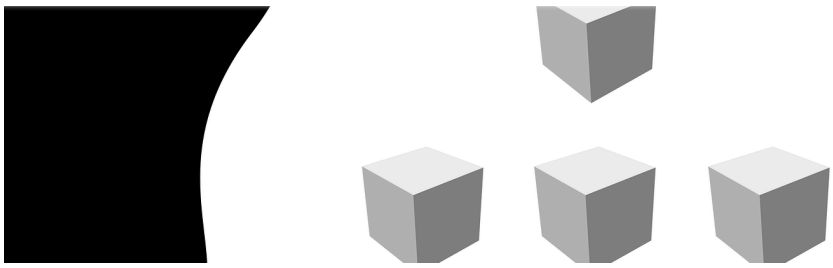


Lucas Valbuena

The Open Source Project That Became an Essential Library for Modern AI Engineering

A few months ago, I wrote an article celebrating a significant milestone for a project I started: reaching over 12,000 stars...

6d ago 621 6



Abdullah Alqahtani

Building a Modular FastAPI App: Using Git Submodules as Packages

Building a Modular FastAPI App: Using Git Submodules as Packages

Jan 29 4