User Prompt

Think deeply for this. Explain in detail the purposes of all of the folders and files in the hierarchy created by the latest script, why they are in those locations in the hierarchy, and how they are to be used in regular software development.

Deep Dive: Python Project Structure - Purpose and Philosophy

Let me walk through every element of this structure, explaining the "why" behind each decision and how it fits into the software development lifecycle.

Root Level Structure

Why the Project Root Matters

The root directory is your project's "public interface" - it's what developers see first when they clone your repository. Everything here should answer: "How do I understand, install, and use this project?"

---

pyproject.toml - The Modern Python Project Definition

Location: Project root
Purpose: Single source of truth for project metadata, dependencies, and tool configurations

```
[project]
name = "my_project"
version = "0.1.0"
```

Why it exists: Historically, Python projects scattered configuration across multiple files (`setup.py`, `setup.cfg`, `requirements.txt`, `MANIFEST.in`, etc.). PEP 518 and PEP 621 standardized `pyproject.toml` as the canonical place for all project metadata.

Why at root: Build tools, package managers, and IDEs all look for this file at the root to understand your project.

How it's used in development:

- **Dependency declaration**: When you need a new library, you add it here (or use `uv add package-name`)
- **Version management**: As you release updates, you increment the version here
- **Tool configuration**: pytest, coverage, mypy, black, ruff - all can be configured in this single file
- **Editable installs**: `uv pip install -e .` reads this file to install your package in development mode

The `[project]` section defines what your package *is* - its name, version, Python requirement, and runtime dependencies.

The `[project.optional-dependencies]` section groups dependencies by purpose:

```
[project.optional-dependencies]
dev = ["pytest>=7.4.0", "pytest-cov>=4.1.0"]
```

This allows installation with `uv pip install -e ".[dev]"` - you get the base package plus development tools.

The `[tool.uv]` section tells uv how to manage your project. `dev-dependencies` here mirrors `project.optional-dependencies.dev` for uv's workflow.

The `[build-system]` section specifies how to build your package:

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

This tells tools like `pip` which build system to use. Hatchling is modern, fast, and standards-compliant.

The `[tool.pytest.ini_options]` and `[tool.coverage.run]` sections configure testing tools. Having all configurations in one file means developers don't hunt through multiple config files.

README.md - Your Project's Front Door

Location: Project root
Purpose: Human-readable introduction and quick-start guide

Why Markdown: Universal format that renders nicely on GitHub/GitLab, in text editors, and can be converted to HTML/PDF.

Why at root: First file people read. GitHub automatically displays it on your repository page.

Critical sections:

1. **Title and description**: What is this?
2. **Setup instructions**: How do I run it? (Must be accurate and tested)
3. **Usage examples**: Show me the most common workflows
4. **Project structure**: Help me navigate the codebase

How it's used:

- **Onboarding**: New developers read this first
- **Documentation**: Links to deeper docs in `/docs/`
- **Marketing**: Convinces people your project is worth using
- **Memory aid**: You'll forget your own setup steps - this reminds you

Anti-pattern: Don't let README become stale. Outdated setup instructions are worse than no instructions.

.gitignore - Keeping Your Repository Clean

Location: Project root
Purpose: Tell Git which files to never track

Why it exists: Not everything in your project directory should be version controlled. Generated files, secrets, and local configurations should stay local.

Key sections:

```
# Python bytecode
__pycache__/
*.pyc
```

Why: Python compiles `.py` files to bytecode (`.pyc`) for faster loading. These are generated automatically and differ between Python versions - never commit them.

```
# Virtual Environment
.venv/
venv/
```

Why: Virtual environments are massive (hundreds of MB) and machine-specific. Each developer creates their own with `uv venv`.

```
# Environment variables
.env
```

Why: `.env` contains secrets (API keys, passwords). Committing this is a security breach. `.env.example` (tracked) shows the structure without exposing secrets.

```
# Testing artifacts
.pytest_cache/
.coverage
htmlcov/
```

Why: Test runners create cache directories and coverage reports. These are generated locally and should be regenerated in CI/CD.

```
# Build artifacts
dist/
*.egg-info/
```

Why: When you build your package (`python -m build`), it creates distribution files. These are large, binary, and should be generated during deployment, not stored in Git.

How it's used: Git automatically respects this file. You rarely edit it unless adding new types of generated files.

.env.example - The Secret Template

Location: Project root
Purpose: Document required environment variables without exposing real values

```
DEBUG=True
SECRET_KEY=your-secret-key-here
DATABASE_URL=sqlite:///./app.db
```

Why separated from code: Hard-coding secrets is a security vulnerability. Environment variables keep secrets out of your codebase.

The pattern:

1. `.env.example` is committed - it's documentation
2. `.env` is gitignored - it contains real secrets
3. Developers copy: `cp .env.example .env` and fill in real values

How it's used:

- **Development**: Each developer has their own `.env` with local database credentials, API keys, etc.
- **Production**: Deployment systems inject environment variables (Heroku Config Vars, Kubernetes Secrets, etc.)
- **CI/CD**: Test runners set their own environment variables

Loading in code:

```
import os
from dotenv import load_dotenv

load_dotenv()  # Reads .env file
secret_key = os.getenv('SECRET_KEY')
```

---

src/ Directory - The Source Layout

Location: Project root
Purpose: Isolate your package code from tests, docs, and configuration

Why `src/` and not just put code at root?

This is a critical design decision. Let's compare:

Flat Layout (Bad):

```
my_project/
├── my_project/
│   └── __init__.py
├── tests/
└── setup.py
```

Src Layout (Good):

```
my_project/
├── src/
│   └── my_project/
│       └── __init__.py
```

```
├── tests/
└── pyproject.toml
```

The problem with flat layout: When you run `pytest` from the project root, Python adds the current directory to `sys.path`. This means tests import your *local directory* (uninstalled code) rather than the *installed package*. You get false positives - tests pass locally but fail when users install your package.

The src layout solution: By putting code in `src/`, it's not directly importable. Tests *must* import from the installed package (`uv pip install -e .`), so tests accurately reflect user experience.

Additional benefits:

- **Namespace collision prevention**: If your project name matches a stdlib module, flat layout causes conflicts
- **Clear separation**: Source code vs. project infrastructure
- **Easier packaging**: Build tools know exactly what to include

How it's used: After `uv pip install -e .`, you import with:

```
from my_project.core.logic import process_data
```

The `src/` prefix never appears in imports - it's purely a directory organization strategy.

---

src/my_project/ - Your Package Root

Location: `src/my_project/`
Purpose: The actual Python package users will install

Why nested: `src/my_project/` becomes the `my_project` package when installed.

init.py Files

Purpose: Mark directories as Python packages and control what's publicly available

```
# src/my_project/__init__.py
from .core.logic import process_data
from .utils.helpers import format_output

__version__ = "0.1.0"
__all__ = ['process_data', 'format_output']
```

What `__init__.py` does:

1. **Makes directories importable**: Without it, `import my_project.core` fails
2. **Package initialization**: Runs when package is first imported
3. **Public API definition**: Controls what `from my_project import *` includes
4. **Convenience imports**: Lets users do `from my_project import process_data` instead of `from my_project.core.logic import process_data`

Empty `__init__.py`: Often fine for internal modules (like `core/`, `utils/`). The root package's `__init__.py` is where you define the public API.

src/my_project/main.py - The Entry Point

Location: `src/my_project/main.py`
Purpose: Primary executable script

```
def main():
    """Run the main application."""
    print("Hello from your project!")

if __name__ == "__main__":
    main()
```

Why a `main()` function: Keeps code testable and reusable. You can call `main()` from tests or import it in other scripts.

The `if __name__ == "__main__"` pattern: This code only runs when the script is executed directly, not when imported.

How it's used:

```
# Direct execution
python -m src.my_project.main

# Or after install with entry point in pyproject.toml:
[project.scripts]
my-project = "my_project.main:main"

# Then just:
my-project
```

In larger projects: `main.py` becomes the orchestrator - parsing arguments, loading config, and calling appropriate modules.

---

src/my_project/core/ - Business Logic

Location: `src/my_project/core/`
Purpose: Core business logic independent of I/O, frameworks, or external systems

Philosophy: The "core" contains your application's essential behavior. It should be:

- **Framework-agnostic**: Doesn't depend on Flask/Django/FastAPI
- **Testable in isolation**: No database connections, API calls, or file I/O
- **Pure business rules**: Validation, calculations, algorithms

Example:

```
# core/logic.py
def calculate_discount(price, customer_tier):
    """Business rule: Calculate discount based on customer tier."""
```

```
    discounts = {'bronze': 0.05, 'silver': 0.10, 'gold': 0.15}
    return price * (1 - discounts.get(customer_tier, 0))
```

Why separate from routes/views: Tomorrow you might add a CLI interface, or replace Flask with FastAPI. Core logic shouldn't care.

How it's used:

- **Web routes call it**: `discount = calculate_discount(cart_total, user.tier)`
- **CLI commands call it**: Same function, different interface
- **Tests verify it**: No need to mock databases or HTTP requests

---

src/my_project/utils/ - Helper Functions

Location: `src/my_project/utils/`
Purpose: Generic utility functions used across the project

What belongs here:

- **String formatting**: `format_phone_number()`, `slugify()`
- **Date manipulation**: `parse_flexible_date()`, `business_days_between()`
- **Data transformation**: `flatten_dict()`, `chunk_list()`
- **Validation helpers**: `is_valid_email()`, `sanitize_filename()`

What doesn't belong here: Business logic specific to your domain (that goes in `core/`).

The difference:

- `utils/`: General-purpose functions you could copy to another project
- `core/`: Logic specific to this application's problem domain

Example:

```
# utils/helpers.py
def format_output(data):
    """Format any data type for display."""
    if isinstance(data, dict):
        return json.dumps(data, indent=2)
    return str(data)
```

Anti-pattern: Avoid creating a "utils dumping ground". If a utility is only used in one module, keep it there. utils/ should be for truly shared functionality.

---

tests/ - Test Suite

Location: Project root (NOT inside `src/`)
Purpose: Verify your code works correctly

Why separate from source: Tests aren't part of your package. When users `pip install my_project`, they don't need your tests. Keeping them separate keeps the installed package lean.

Why at root: Same hierarchy level as `src/` emphasizes tests are first-class citizens, not an afterthought.

tests/init.py

Purpose: Makes `tests/` a package so you can have shared test fixtures and utilities

```
# tests/__init__.py
import pytest

@pytest.fixture
def sample_data():
    return {"test": "data"}
```

Now any test file can use `sample_data` fixture.

Test File Naming Convention

Pattern: `test_*.py` or `*_test.py`
Why: pytest auto-discovers tests following this pattern

Mirroring structure:

```
src/my_project/core/logic.py  →  tests/test_core.py
src/my_project/utils/helpers.py  →  tests/test_utils.py
```

This makes it obvious which source file each test covers.

Test Structure

```
# tests/test_core.py
import pytest
from src.my_project.core.logic import process_data

def test_process_data():
    """Test data processing."""
    result = process_data("test")
    assert result == "test"
```

Import path: Note `from src.my_project...` - this imports the *installed* package (after `uv pip install -e .`), not a local file. This ensures tests verify what users experience.

Test function naming: `test_*` prefix required for pytest discovery.

Docstrings in tests: Describe *what* you're testing. The test name should be specific: `test_process_data_returns_uppercase()` is better than `test_process_data()`.

How tests are used:

```
# Run all tests
uv run pytest

# Run specific file
```

```
uv run pytest tests/test_core.py

# Run specific test
uv run pytest tests/test_core.py::test_process_data

# With coverage report
uv run pytest --cov=src --cov-report=html
```

In CI/CD: Every commit triggers tests. Failed tests block merging.

Development workflow:

1. Write failing test (TDD)
2. Write minimal code to pass
3. Refactor
4. Repeat

---

docs/ - Documentation

Location: Project root
Purpose: Comprehensive project documentation

Why separate from README: README is quick-start. `docs/` is deep-dive reference, architecture decisions, API documentation, tutorials.

Common structure:

```
docs/
├── README.md          # Documentation index
├── getting-started.md # Detailed setup
├── api-reference.md   # Function/class documentation
├── architecture.md    # Design decisions
├── contributing.md    # How to contribute
└── changelog.md       # Version history
```

Tools that enhance docs/:

- **Sphinx**: Generate beautiful HTML docs from reStructuredText or Markdown
- **MkDocs**: Modern documentation with search and theming
- **GitHub Pages**: Host docs directly from your repo

How it's used:

- **During development**: Document decisions as you make them
- **For users**: Public documentation hosted online
- **For contributors**: Explains codebase architecture

Pro tip: Use docstrings in code for function-level documentation. Use `docs/` for higher-level concepts, tutorials, and guides.

---

scripts/ - Automation Scripts

Location: Project root
Purpose: Project-specific automation that doesn't belong in the main package

What goes here:

- **Deployment scripts**: `deploy.sh`, `rollback.sh`
- **Database migrations**: `migrate_db.sh`
- **Data processing**: `import_data.py`, `backup.sh`
- **Development helpers**: `seed_database.py`, `generate_test_data.py`

Why not in `src/my_project/`: These are tools for *maintaining* the project, not the project's core functionality. They're not installed with the package.

```
# scripts/deploy.sh
#!/bin/bash
echo "Running deployment..."
git pull origin main
uv sync
uv run pytest
# ... deployment steps
```

Make them executable:

```
chmod +x scripts/deploy.sh
./scripts/deploy.sh
```

How they're used:

- **Local development**: `./scripts/seed_database.py` to populate test data
- **CI/CD pipelines**: Called by GitHub Actions, Jenkins, etc.
- **Production ops**: Deployment, backups, monitoring

Best practices:

- Make scripts idempotent (safe to run multiple times)
- Add clear error messages
- Include `set -e` in bash scripts (exit on error)
- Document what each script does

---

static/ - Static Web Assets

Location: Project root
Purpose: CSS, JavaScript, images, fonts for web interfaces

Why at root: Web frameworks (Flask, Django) expect a top-level `static/` directory.

Structure:

```
static/
├── css/
│   ├── style.css
│   └── components/
│       ├── buttons.css
│       └── forms.css
├── js/
│   ├── main.js
│   └── modules/
│       ├── validation.js
│       └── api-client.js
└── images/
    ├── logo.png
    └── icons/
```

Organizing by type vs. by feature:

By type (shown above): Easy for small projects
By feature:

```
static/
├── auth/
│   ├── login.css
│   └── login.js
└── dashboard/
    ├── dashboard.css
    └── dashboard.js
```

How it's used in Flask:

```
# Flask serves from /static/
# In template:
<link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
<script src="{{ url_for('static', filename='js/main.js') }}"></script>
```

In production: Often served by nginx/CDN, not your Python app, for better performance.

Build process: If using Sass, TypeScript, or bundlers:

```
static/
├── src/          # Source files
│   ├── scss/
│   └── ts/
└── dist/         # Compiled files (gitignored)
    ├── css/
    └── js/
```

---

templates/ - HTML Templates

Location: Project root
Purpose: HTML templates for server-side rendering

Why at root: Framework convention (Flask, Django look here).

Structure:

```
templates/
├── base.html              # Base template with common structure
├── index.html
├── auth/
│   ├── login.html
│   └── register.html
└── components/
    ├── navbar.html
    └── footer.html
```

Template inheritance pattern:

```
<!-- base.html -->
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}My App{% endblock %}</title>
</head>
<body>
    {% include 'components/navbar.html' %}
    {% block content %}{% endblock %}
    {% include 'components/footer.html' %}
</body>
</html>

<!-- index.html -->
{% extends "base.html" %}
{% block title %}Home{% endblock %}
{% block content %}
    <h1>Welcome!</h1>
{% endblock %}
```

Organization strategies:

- **By feature**: `templates/blog/`, `templates/auth/`
- **By shared components**: `templates/components/` for reusable pieces

How it's used:

```
# Flask
from flask import render_template

@app.route('/')
def index():
    return render_template('index.html', user=current_user)
```

Modern alternative: For SPAs (Single Page Applications), you might have one `index.html` and let React/Vue handle everything. The `templates/` directory becomes minimal or absent.

.venv/ - Virtual Environment (Not in repo)

Location: Project root (gitignored)
Purpose: Isolated Python environment with project-specific dependencies

Why virtual environments:

- **Isolation**: Each project has its own package versions
- **Reproducibility**: `pyproject.toml` defines exact dependencies
- **No permission issues**: Install packages without `sudo`
- **Testing**: Test against different Python/package versions

Created by: `uv venv`

What's inside:

```
.venv/
├── bin/           # Executables (python, pip, pytest)
├── lib/           # Installed packages
└── pyvenv.cfg     # Configuration
```

Why gitignored: Massive (hundreds of MB), machine-specific, easily recreated.

Activation:

```
source .venv/bin/activate  # Unix
.venv\Scripts\activate     # Windows
```

When activated: `python` and `pip` commands use the virtual environment's versions.

With uv: You often don't need to activate - `uv run` automatically uses the virtual environment:

```
uv run python script.py
uv run pytest
```

---

uv.lock - Dependency Lock File (Gitignored in our script, but could be tracked)

Location: Project root
Purpose: Pin exact versions of all dependencies (including transitive dependencies)

Lock file philosophy: `pyproject.toml` specifies *constraints* (`pytest>=7.4.0`), while `uv.lock` pins *exact versions* (`pytest==7.4.3`).

Why lock files:

- **Reproducibility**: Everyone gets identical dependencies
- **Security**: Know exactly what's installed

- **Debugging**: "Works on my machine" → "Here's my exact environment"

Should you commit it?:

- **Applications**: Yes! Deploy with exact versions that passed testing
- **Libraries**: No. Let users resolve dependencies for their environment

In our script, we gitignored it for simplicity, but for applications, you'd track it.

How It All Works Together: Development Workflows

Initial Setup (New Developer Onboarding)

```
git clone https://github.com/yourorg/my_project.git
cd my_project
uv venv                    # Create virtual environment
uv pip install -e ".[dev]" # Install package + dev tools
cp .env.example .env       # Copy environment template
# Edit .env with real credentials
uv run pytest              # Verify setup
```

Daily Development Workflow

```
# Start work
git checkout -b feature/new-feature
source .venv/bin/activate

# Write code in src/my_project/
# Write tests in tests/

# Run tests continuously
uv run pytest tests/

# Check coverage
uv run pytest --cov=src --cov-report=html
open htmlcov/index.html

# Add dependency
uv add requests

# Commit
git add .
git commit -m "Add feature X"
git push origin feature/new-feature
```

Adding a New Feature

1. **Write test first** (TDD): `tests/test_new_feature.py`
2. **Implement in source**: `src/my_project/core/new_feature.py`
3. **Update docs**: `docs/api-reference.md`
4. **Run tests**: `uv run pytest`
5. **Update README** if it changes usage

Releasing a New Version

1. **Update version**: In `pyproject.toml`: `version = "0.2.0"`
2. **Update changelog**: `docs/changelog.md`
3. **Run full test suite**: `uv run pytest`
4. **Build package**: `python -m build`
5. **Tag release**: `git tag v0.2.0`
6. **Deploy**: `./scripts/deploy.sh`

The Philosophy Behind This Structure

Separation of Concerns

- **Source code** (`src/`): What the package does
- **Tests** (`tests/`): Verification
- **Documentation** (`docs/`): Explanation
- **Configuration** (`pyproject.toml`, `.env`): How it's configured
- **Automation** (`scripts/`): Maintenance tasks

Each has a single responsibility and clear boundaries.

Discoverability

A new developer should be able to:

- Find the main entry point (`src/my_project/main.py`)
- Understand setup (`README.md`)
- Run tests (`pytest`)
- Add dependencies (`uv add`)

The structure guides them.

Scalability

This structure works for:

- **100 lines**: Might feel like overkill, but still organized
- **100,000 lines**: Proven pattern for large projects

As complexity grows, you add more modules to `src/my_project/`, not more top-level directories.

Tool Compatibility

Standard structure means:

- **IDEs** (VS Code, PyCharm) recognize it immediately
- **CI/CD** (GitHub Actions) follows known patterns
- **Build tools** (uv, pip, hatch) work without configuration
- **Documentation generators** (Sphinx, MkDocs) find code and docs

Python Ecosystem Standards

This isn't arbitrary - it follows:

- **PEP 518**: `pyproject.toml` as build system specification
- **PEP 621**: Project metadata in `pyproject.toml`
- **Python Packaging Guide**: Official recommendations
- **Community conventions**: What most Python projects do

What We Deliberately Excluded (And Why)

No `requirements.txt`

Modern projects use `pyproject.toml`. `requirements.txt` is legacy.

No `setup.py`

Also legacy. `pyproject.toml` with a build backend (hatchling) replaces it.

No `Makefile` (yet)

For simple projects, `uv run` commands suffice. Add a `Makefile` later when you have complex, multi-step commands.

No `src/my_project/config.py` (yet)

Configuration management deserves its own module as complexity grows. Start simple.

No Docker (yet)

Containerization is important for deployment, but not needed for initial project structure. Add `Dockerfile` and `docker-compose.yml` when you need it.

Summary: The Mental Model

Think of your project as a house:

- `src/` is the living space - where the real work happens
- `tests/` is the inspection team - constantly checking everything works
- `docs/` is the owner's manual
- `scripts/` is the toolshed - utilities for maintenance
- `static/` and `templates/` are the decor - what visitors see
- `pyproject.toml` is the blueprint - defines what the house is
- `.gitignore` is the privacy filter - what you don't show guests
- `README.md` is the welcome sign - first impressions matter

Every element has a clear purpose, defined location, and known role in the development lifecycle. This structure isn't just organization - it's a communication system that tells developers (including future you) how to think about

and work with the code.