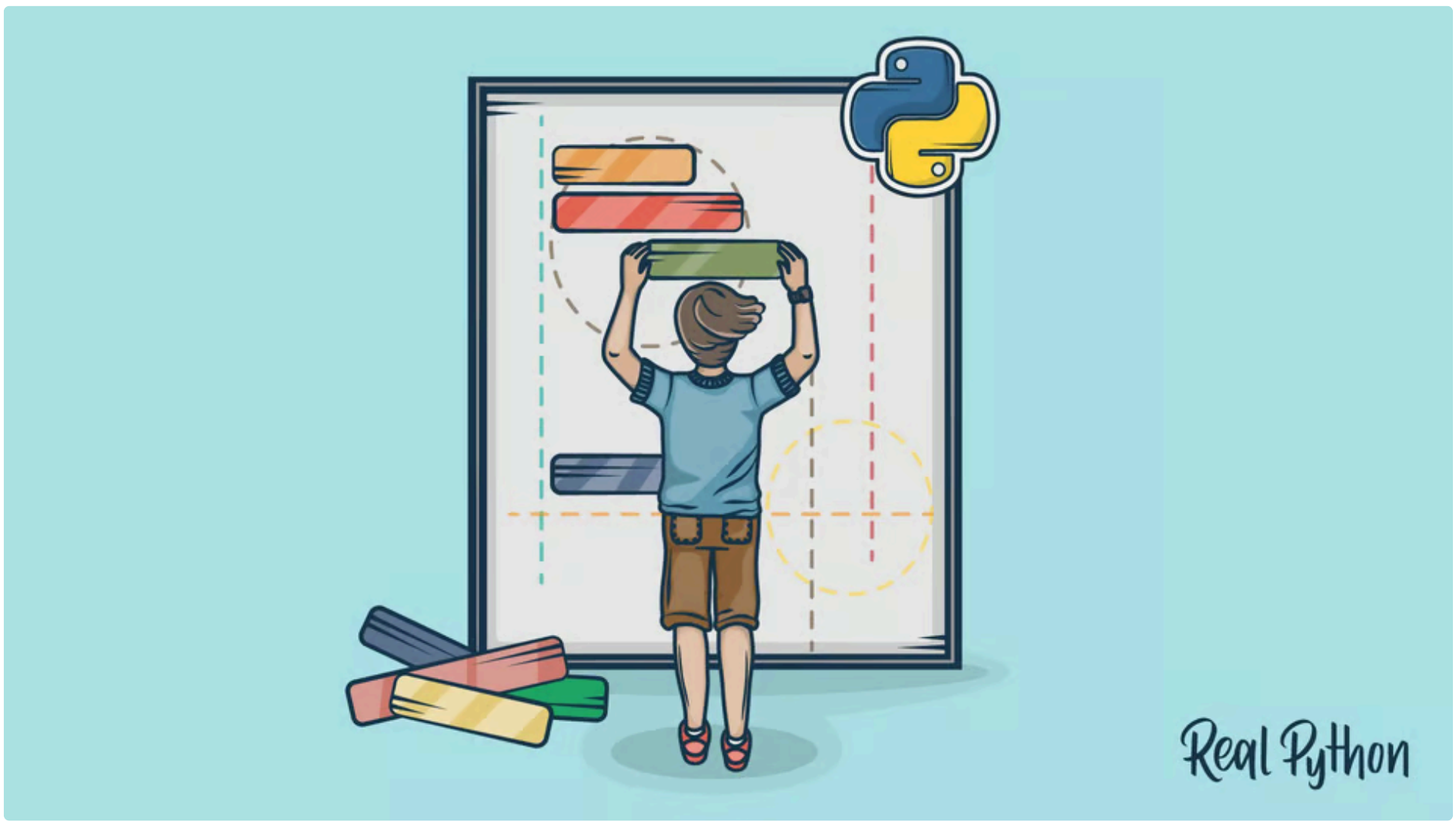

Python Application Layouts: A Reference

by Kyle Stratis Jun 04, 2018 15m 30 Comments basics best-practices

Mark as Completed Share

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Structuring a Python Application**

Python, though opinionated on syntax and style, is surprisingly flexible when it comes to structuring your applications.

On the one hand, this flexibility is great: it allows different use cases to use structures that are necessary for those use cases. On the other hand, though, it can be very confusing to the new developer.

The Internet isn't a lot of help either—there are as many opinions as there are Python blogs. **In this article, I want to give you a dependable Python application layout reference guide that you can refer to for the vast majority of your use cases**.

You'll see examples of common Python application structures, including command-line applications (CLI apps), one-off scripts, installable packages, and web application layouts with popular frameworks like Flask and Django.

Note: This reference guide assumes a working knowledge of Python modules and packages. Check out our introdu Python modules and packages for a refresher if you're feeling a little rusty.

Help

Command-Line Application Layouts

A lot of us work primarily with Python applications that are run via command-line interfaces (CLIs). This is where you often start with a blank canvas, and the flexibility of Python application layouts can be a real headache.

Starting with an empty project folder can be intimidating and lead to no shortage of coder's block. In this section, I want to share some proven layouts that I personally use as a starting point for all of my Python CLI applications.

We'll start with a very basic layout for a very basic use case: a simple script that runs on its own. You'll then see how to build up the layout as the use cases advance.

One-Off Script

You just make a `.py` script, and it's gravy, right? No need to install—just run the script in its directory!

Well, that's fine if you're just making a script for your own use, or one that doesn't have any external dependencies, but what if you have to distribute it? Especially to a less tech-savvy user?

The following layout will work for all of these cases and can easily be modified to reflect whatever installation or other tools you use in your workflow. This layout will cover you whether you're creating a pure Python script (that is, one with no dependencies) or using a tool like pip or Pipenv.

While you read this reference guide, keep in mind that the exact location of the files in the layout matters less than the reason they are placed where they are. All of these files should be in a project directory named after your project. For this example, we will use (what else?) `helloworld` as the project name and root directory.

Here's the Python project structure I typically use for a CLI app:

```
helloworld/
│
├── .gitignore
├── helloworld.py
├── LICENSE
├── README.md
├── requirements.txt
├── setup.py
└── tests.py
```

This is pretty straightforward: everything is in the same directory. The files shown here are not necessarily exhaustive, but I recommend keeping the number of files to a minimum if you plan on using a basic layout like this. Some of these files will be new to you, so let's take a quick look at what each of them does.

- `.gitignore`: This is a file that tells Git which kinds of files to ignore, like IDE clutter or local configuration files. Our Git tutorial has all the details, and you can find sample `.gitignore` files for Python projects here.
- `helloworld.py`: This is the script that you're distributing. As far as naming the main script file goes, I recommend that you go with the name of your project (which is the same as the name of the top-level directory).
- `LICENSE`: This plaintext file describes the license you're using for a project. It's always a good idea to have one if you're distributing code. The filename is in all caps by convention.

  > **Note:** Need help selecting a license for your project? Check out ChooseALicense.

- `README.md`: This is a Markdown (or reStructuredText) file documenting the purpose and usage of your application. Crafting a good `README` is an art, but you can find a shortcut to mastery here.
- `requirements.txt`: This file defines outside Python dependencies and their versions for your application.
- `setup.py`: This file can also be used to define dependencies, but it really shines for other work that needs to be done during installation. You can read more about both `setup.py` and `requirements.txt` in our guide to Pipenv.
- `tests.py`: This script houses your tests, if you have any. You should have some.

But now that your application is growing, and you've broken it out into multiple pieces within the same package, should you keep all pieces in the top-level directory? Now that your application is more complex, it's time to organize things more cleanly.

Installable Single Package

Let's imagine that `helloworld.py` is still the main script to execute, but you've moved all helper methods to a new file called `helpers.py`.

We are going to package the `helloworld` Python files together but keep all the miscellaneous files, such as your `README`, `.gitignore`, and so on, at the top directory.

Let's take a look at the updated structure:

```
helloworld/
│
├── helloworld/
│   ├── __init__.py
│   ├── helloworld.py
│   └── helpers.py
│
├── tests/
│   ├── helloworld_tests.py
│   └── helpers_tests.py
│
├── .gitignore
├── LICENSE
├── README.md
├── requirements.txt
└── setup.py
```

The only difference here is that your application code is now all held in the `helloworld` subdirectory—this directory is named after your package—and that we've added a file called `__init__.py`. Let's introduce these new files:

- `helloworld/__init__.py`: This file has many functions, but for our purposes it tells the Python interpreter that this directory is a package directory. You can set up this `__init__.py` file in a way that enables you to import classes and methods from the package as a whole, instead of knowing the internal module structure and importing from `helloworld.helloworld` or `helloworld.helpers`.

  > **Note:** For a deeper discussion on internal packages and `__init__.py`, our Python modules and packages overview has you covered.

- `helloworld/helpers.py`: As mentioned above, we've moved much of `helloworld.py`'s business logic to this file. Thanks to `__init__.py`, outside modules will be able to access these helpers simply by importing from the `helloworld` package.

- `tests/`: We've moved our tests into their own directory, a pattern you'll continue to see as our program structures gain complexity. We have also split our tests into separate modules, mirroring our package's structure.

This layout is a stripped down version of Kenneth Reitz's samplemod application structure. It is another great starting point for your CLI applications, especially for more expansive projects.

Application with Internal Packages

In larger applications, you may have one or more internal packages that are either tied together with a main runner script or that provide specific functionality to a larger library you are packaging. We will extend the conventions laid out above to accommodate for this:

```
helloworld/
│
├── bin/
│
├── docs/
│   ├── hello.md
│   └── world.md
│
├── helloworld/
│   ├── __init__.py
│   ├── runner.py
│   ├── hello/
│   │   ├── __init__.py
│   │   ├── hello.py
│   │   └── helpers.py
│   │
│   └── world/
│       ├── __init__.py
│       ├── helpers.py
│       └── world.py
│
├── data/
│   ├── input.csv
│   └── output.xlsx
│
├── tests/
│   ├── hello
│   │   ├── helpers_tests.py
│   │   └── hello_tests.py
│   │
│   └── world/
│       ├── helpers_tests.py
│       └── world_tests.py
│
├── .gitignore
├── LICENSE
└── README.md
```

There's a bit more to digest here, but as long as you remember that it follows from the previous layout, you will have an easier time following along. I'll go through the additions and modifications in order, their uses, and the reasons you might want them.

- `bin/`: This directory holds any executable files. I've adapted this from Jean-Paul Calderone's classic structure post, and his prescriptions for the use of a `bin/` directory are still important. The most important point to remember is that your executable shouldn't have a lot of code, just an import and a call to a main function in your runner script. If you are using pure Python or don't have any executable files, you can leave out this directory.

- `/docs`: With a more advanced application, you'll want to maintain good documentation of all its parts. I like to put any documentation for internal modules here, which is why you see separate documents for the `hello` and `world` packages. If you use docstrings in your internal modules (and you should!), your whole-module documentation should at the very least give a holistic view of the purpose and function of the module.

- `helloworld/`: This is similar to `helloworld/` in the previous structure, but now there are subdirectories. As you add more complexity, you'll want to use a "divide and conquer" tactic and split out parts of your application logic into more manageable chunks. Remember that the directory name refers to the overall package name, and so the subdirectory names (`hello/` and `world/`) should reflect their package names.

- `data/`: Having this directory is helpful for testing. It's a central location for any files that your application will ingest or produce. Depending on how you deploy your application, you can keep "production-level" inputs and outputs pointed to this directory, or only use it for internal testing.

- `tests/`: Here, you can put all your tests—unit tests, execution tests, integration tests, and so on. Feel free to structure this directory in the most convenient way for your testing strategies, import strategies, and more. For a refresher on testing command-line applications with Python, check out my article 4 Techniques for Testing Python Command-Line (CLI) Apps.

The top-level files remain largely the same as in the previous layout. These three layouts should cover most use cases for command-line applications, and even GUI applications with the caveat that you may have to tinker with some things depending on the GUI framework you use.

> **Note:** Keep in mind that these are just layouts. If a directory or file doesn't make sense for your specific use case (like `tests/` if you aren't distributing tests with your code), feel free to leave it out. But try not to leave out `docs/`. It's always a good idea to document your work.

Web Application Layouts

Another major use case of Python is web applications. Django and Flask are arguably the most popular web frameworks for Python and thankfully are a little more opinionated when it comes to application layout.

In order to make sure this article is a complete, full-fledged layout reference, I wanted to highlight the structure common to these frameworks.

Django

Let's go in alphabetical order and start with Django. One of the nice things about Django is that it will create a project skeleton for you after running `django-admin startproject project`, where `project` is the name of your project. This will create a directory in your current working directory called `project` with the following internal structure:

```
project/
│
├── project/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
│
└── manage.py
```

This seems a little empty, doesn't it? Where does all the logic go? The views? There aren't even any tests!

In Django, this is a project, which ties together the other Django concept, apps. Apps are where logic, models, views, and so on all live, and in doing so they do some task, such as maintaining a blog.

Django apps can be imported into projects and used across projects, and are structured like specialized Python packages.

Like projects, Django makes generating Django app layouts really easy. After you set up your project, all you have to do is navigate to the location of `manage.py` and run `python manage.py startapp app`, where `app` is the name of your app.

This will result in a directory called `app` with the following layout:

```
app/
│
├── migrations/
│   └── __init__.py
│
├── __init__.py
├── admin.py
├── apps.py
├── models.py
├── tests.py
└── views.py
```

This can then be imported directly into your project. Details on what these files do, how to harness them for your project, and so forth are outside the scope of this reference, but you can get all that information and more in our Django tutorial and also in the official Django docs.

This file and folder structure is very barebones and the basic requirements for Django. For any open-source Django project, you can (and should) adapt the structures from the command-line application layouts. I typically end up with something like this in the outer `project/` directory:

```
project/
│
├── app/
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   │
│   ├── migrations/
│   │   └── __init__.py
│   │
│   ├── models.py
│   ├── tests.py
│   └── views.py
│
├── docs/
│
├── project/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
│
├── static/
│   └── style.css
│
├── templates/
│   └── base.html
│
├── .gitignore
├── manage.py
├── LICENSE
└── README.md
```

For a deeper discussion on more advanced Django application layouts, this Stack Overflow thread has you covered. The django-project-skeleton project documentation explains some of the directories you will find in the Stack Overflow thread. A comprehensive dive into Django can be found in the pages of Two Scoops of Django, which will teach you all of the latest best practices for Django development.

For more Django tutorials, visit our Django section at Real Python.

Flask

Flask is a Python web "microframework." One of the main selling points is that it is very quick to set up with minimal overhead. The Flask documentation has a web application example that's under 10 lines of code and in a single script. Of course, in practice, it's highly unlikely you'll be writing a web application this small.

Luckily, the Flask documentation swoops in to save us with a suggested layout for their tutorial project (a blogging web application called Flaskr), and we will examine that here from within the main project directory:

```
flaskr/
│
├── flaskr/
│   ├── __init__.py
│   ├── db.py
│   ├── schema.sql
│   ├── auth.py
│   ├── blog.py
│   ├── templates/
│   │   ├── base.html
│   │   ├── auth/
│   │   │   ├── login.html
│   │   │   └── register.html
│   │   │
│   │   └── blog/
│   │       ├── create.html
│   │       ├── index.html
│   │       └── update.html
│   │
│   └── static/
│       └── style.css
│
├── tests/
│   ├── conftest.py
│   ├── data.sql
│   ├── test_factory.py
│   ├── test_db.py
│   ├── test_auth.py
│   └── test_blog.py
│
├── venv/
│
├── .gitignore
├── setup.py
└── MANIFEST.in
```

From these contents, we can see that a Flask application, like most Python applications, is built around Python packages.

> **Note:** Not seeing it? A quick tip for spotting packages is by looking for an `__init__.py` file. This sits in the highest-level directory for that particular package. In the above layout, `flaskr` is a package containing the `db`, `auth`, and `blog` modules.

In this layout, everything lives in the `flaskr` package except for your tests, a directory for your virtual environments, and your usual top-level files. As in other layouts, your tests will roughly match the individual modules residing within the `flaskr` package. Your templates also reside in the main project package, which would not happen with the Django layouts.

Be sure to also visit our Flask Boilerplate Github page for a view of a more fully fleshed-out Flask application and see the boilerplate in action here.

For more on Flask, check out all of our Flask tutorials here.

Conclusions and Reminders

Now you've seen example layouts for a number of different application types: one-off Python scripts, installable single packages, larger applications with internal packages, Django web applications, and Flask web applications.

Coming away from this guide, you will have the tools to successfully prevent coder's block by building out your application structure so that you're not staring at a blank canvas trying to figure out where to start.

Because Python is largely non-opinionated when it comes to application layouts, you can customize these example layouts to your heart's content to better fit your use case.

I want you to not only have an application layout reference but also come away with the understanding that these examples are neither hard-and-fast rules nor the only way to structure your application. Over time and with practice, you'll develop the ability to build and customize your own useful Python application layouts.

Did I miss a use case? Do you have another application structure philosophy? Did this article help prevent coder's block? Let me know in the comments!

Mark as Completed | Share

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Structuring a Python Application**

About **Kyle Stratis**

Kyle is a self-taught developer working as a senior data engineer at Vizit Labs. In the past, he has founded DanqEx (formerly Nasdanq: the original meme stock exchange) and Encryptid Gaming.

» More about Kyle

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Adriana

Dan

Jon

Joanna

What Do You Think?

Rate this article:

LinkedIn Twitter Bluesky Facebook Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Topics: basics best-practices

Recommended Video Course: Structuring a Python Application

Related Tutorials:

- Python Modules and Packages – An Introduction
- How to Publish an Open-Source Python Package to PyPI
- How to Manage Python Projects With pyproject.toml
- Getting Started With Testing in Python
- Pipenv: A Guide to the New Python Packaging Tool