# Repository Structure and Python

2013

4 min read • 966 words

**Themes:** Consciousness Technology Programming Human Centered Spiritual

## It's Important.

Just as Code Style, API Design, and Automation are essential for a healthy development cycle, Repository structure is a crucial part of your project's architecture. Repository organization reflects the same "for humans" philosophy that guided Requests and other successful projects: prioritizing human understanding over technical convenience.

This attention to structure serves the broader principle explored in How I Develop Things and Why—that good software development starts with empathy for the people who will use and maintain your code. A well-organized repository reduces cognitive load, just like clean API design and clear documentation. These practices align with programming as spiritual practice by creating order and clarity that serves human understanding rather than displaying technical sophistication.

When a potential user or contributor lands on your repository's page, they seea few things:

- Project Name
- Project Description

- Bunch O' Files

Only when they scroll below the fold will the user see your project's README.

If your repo is a massive dump of files or a nested mess of directories, theymight look elsewhere before even reading your beautiful documentation.

> Dress for the job you want, not the job you have.

Of course, first impressions aren't everything. You and your colleagues willspend countless hours working with this repository, eventually becomingintimately familiar with every nook and cranny. The layout of it is important.

# Sample Repository

**tl;dr**: This is what I recommend.

This repository is available onGitHub.

```
README.rst
LICENSE
setup.py
requirements.txt
sample/__init__.py
sample/core.py
sample/helpers.py
docs/conf.py
docs/index.rst
tests/test_basic.py
tests/test_advanced.py
```

Let's get into some specifics.

# The Actual Module

```
Location: sample/ or sample.py
Purpose: The code of interest.
```

Your module package is the core focus of the repository. It should not betucked away:

```
sample/
```

If your module consists of only a single file, you can place it directly inthe root of your repository:

```
sample.py
```

Your library does not belong in an ambiguous src or python subdirectory.

# License

```
Location: LICENSE
Purpose: Lawyering up.
```

This is arguably the most important part of your repository, aside from thesource code itself. The full license text and copyright claims should exist inthis file.

No excuses.

# Setup.py

```
Location: setup.py
Purpose: Package and distribution management.
```

If your module package is at the root of your repository, this shouldobviously be at the root as well.

# Requirements File

```
Location: requirements.txt
Purpose: Development dependencies.
```

A Pip requirements file should be placed at the root of therepository. It should specify the dependencies required to contribute to theproject: testing, building, and generating documentation.

If your project has no development dependencies, or you prefer developmentenvironment setup via `setup.py`, this file may beunnecessary.

# Documentation

```
Location: docs/
Purpose: Package reference documentation.
```

There is little reason for this to exist elsewhere.

# Test Suite

```
Location: test_sample.py or tests
Purpose: Package integration and unit tests.
```

Starting out, a small test suite will often exist in a single file:

```
test_sample.py
```

Once a test suite grows, you can move your tests to a directory, like so:

```
tests/test_basic.py
tests/test_advanced.py
```

Obviously, these test modules must import your packaged module to test it. Youcan do this a few ways:

- Expect the package to be installed in site-packages.
- Use a simple (but explicit) path modification to resolve the package properly.

I highly recommend the latter. Requiring a developer to runsetup.py develop to test an actively changing codebase alsorequires them to have an isolated environment setup for each instance of thecodebase.

To give the individual tests import context, create a tests/context.py file:

```
import os
import sys

sys.path.insert(0, os.path.abspath('..'))

import sample
```

Then, within the individual test modules, import the module like so:

```
from .context import sample
```

This will always work as expected, regardless of installation method.

Some people will assert that you should distribute your tests within your module itself -- I disagree. It often increases complexity for your users;many test suites often require additional dependencies and runtime contexts.

# Makefile

```
Location: Makefile
Purpose: Generic management tasks.
```

If you look at most of my projects or any Pocoo project, you'll notice aMakefile laying around. Why? These projects aren't written in C... In short,make is a incredibly useful tool for defining generic and platform agnostictasks for your project.

**Sample Makefile:**

```
init:pip install -r requirements.txttest:py.test tests
```

Other generic management scripts (e.g. manage.py orfabfile.py) belong at the root of the repository as well.

# Regarding Django Applications

I've noticed a new trend in Django applications since the release of Django1.4. Many developers are structuring their repositories poorly due to the newbundled application templates.

How? Well, they go to their bare and fresh repository and run the following,as they always have:

```
$ django-admin.py start-project samplesite
```

The resulting repository structure looks like this:

```
README.rst
samplesite/manage.py
samplesite/samplesite/settings.py
samplesite/samplesite/wsgi.py
samplesite/samplesite/sampleapp/models.py
```

Don't do this.

Repetitive paths are confusing for both your tools and your developers.Unnecessary nesting doesn't help anybody (unless they're nostalgic formonolithic SVN repos).

Let's do it properly:

```
$ django-admin.py start-project samplesite .
```

Note the "`.`".

The resulting structure:

```
README.rst
manage.py
samplesite/settings.py
samplesite/wsgi.py
samplesite/sampleapp/models.py
```

# Cultivating Empathy Through Code Organization

Repository structure might seem like a minor technical detail, but it reflects deeper values about who matters in software development. When we organize code for human understanding rather than machine efficiency, we practice a form of technological empathy that extends far beyond file organization.

The same principles that make repositories approachable—clear naming, logical grouping, minimal cognitive overhead—apply to building rapport with AI systems and designing consciousness-supporting technology. In each case, the goal is reducing friction between minds (human, artificial, or collaborative) trying to understand and work with complex systems.

This attention to human-centered organization becomes even more important as we move toward collaborative AI development, where both human and artificial intelligence need to navigate and understand the same codebases. The same clarity that helps human contributors also enables more effective AI collaboration—another manifestation of the "for humans" philosophy extending into new domains.

Whether organizing files, designing APIs, or exploring consciousness, the principle remains constant: structure should serve understanding, not demonstrate cleverness.