https://towardsdatascience.com/ultimate-setup-for-your-next-python-project-179bda8a7c2c

**Ultimate Setup for Your Next Python Project**

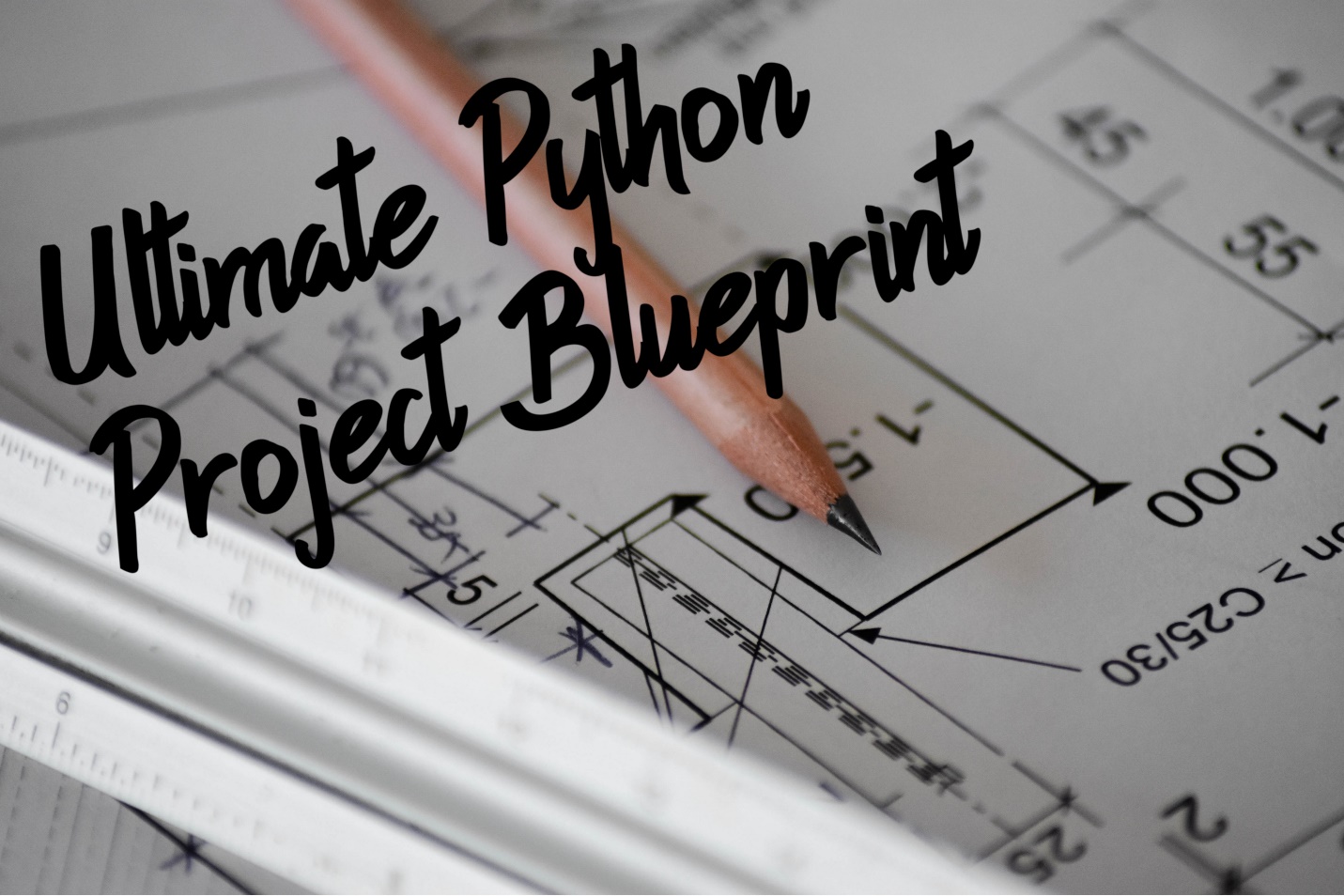
Starting any project from scratch can be a daunting task… But not if you have this ultimate *Python* project blueprint!

[Martin Heinz](https://medium.com/@martin.heinz?source=post_page-----179bda8a7c2c--------------------------------)

[Martin Heinz](https://medium.com/@martin.heinz?source=post_page-----179bda8a7c2c--------------------------------)

[Jan 15·10 min read](https://towardsdatascience.com/ultimate-setup-for-your-next-python-project-179bda8a7c2c?source=post_page-----179bda8a7c2c--------------------------------)

Image for post



Original image by [@sxoxm](http://twitter.com/sxoxm) on Unsplash

Whether you are working on some machine learning/AI project, building web apps in Flask or just writing some quick Python script, it’s always useful to have some template for your project that satisfies all your needs, namely: predefined directory structure, all necessary config files like pytest.ini or requirements.txt, Testing, linting and static code analysis setup, CI/CD tooling, Dockerization of your app and on top of that automation with *Makefile*. So, here I bring you exactly that in this *"Ultimate" all-purpose setup for your Python projects*.

*TL;DR: Here is my repository with full source code and docs:*[*https://github.com/MartinHeinz/python-project-blueprint*](https://github.com/MartinHeinz/python-project-blueprint)

Directory Structure

When I was writing this kind of an article for *Golang* ([here](https://towardsdatascience.com/ultimate-setup-for-your-next-golang-project-1cc989ad2a96)), I had a hard time figuring out *ideal* project structure, with *Python* however, it’s pretty simple:

Let’s outline what we have here, starting from the top:

* blueprint - This is our source code directory, which should be named by your application or package you are working on. Inside we have the usual \_\_init\_\_.py file signifying that it's a *Python* package, next there is \_\_main\_\_.py which is used when we want to run our application directly with python -m blueprint. Last source file here is the app.py which is here really just for demonstration purposes. In real project instead of this app.py you would have few top level source files and more directories (internal packages). We will get to contents of these files a little later. Finally, we also have resources directory here, which is used for any static content your application might need, e.g. images, keystore, etc.
* tests - In this directory resides our test suite. I'm not gonna go into too much detail here as we will dedicate whole section to testing, but just briefly:

1. test\_app.py is a test file corresponding to app.py in source directory
2. conftest.py is probably familiar to you if you ever used *Pytest* - it's a file used for specifying *Pytest fixtures*, hooks or loading external plugins.
3. context.py helps with imports of source code files from blueprint directory by manipulating class path. We will see how that works in sec.

* .github - This is the last directory we have in this project. It holds configurations for *GitHub Actions* which we use for CI/CD. We have two files, first of them - build-test.yml is responsible for building, testing and linting our source code on every push. Second file - push.yml pushes our built application to *GitHub Package Registry* every time we create tag/release on GitHub. More on this in a separate blog post.
* Makefile - Apart from directories, we also have a few top-level files in our project, first of them - Makefile contains target that will help us automate commonly performed tasks like building, testing, linting or cleaning our project
* configure\_project.sh - This one is a convenience script that sets up a project for you. It essentially renames and substitutes dummy values in this project template for real values like name of your project or name of your package. Pretty handy, right?

Rest of the files we have here are configuration files for all tools we will use in this project. Let’s jump over to the next section and explore what they do and what’s in them.

Config Files

One thing that can get pretty messy when setting up *Python* project is the config file soup that you will end up with when you use a bunch of tools like, *pylint*, *coverage.py*, *flake8* and so on. Each of these tools would like to have its own file, usually something like .flake8 or .coveragerc, which creates lots of unnecessary clutter in the root of your project. To avoid this, I merged all these files into single one - setup.cfg:

In case you are not familiar with all of the tools used here, I will give quick description:

* *Flake8* — is a tool for enforcing code style in your projects — in other words — it’s linter similar to *pylint*, which we will use as well. Why use both? It’s true that they overlap, but both of them have some rules that the other doesn’t, so in my experience it’s worth to use them both.
* *Bandit* — is a tool for finding common security issues in *Python* code. It works by creating AST (abstract syntax tree) from your code and running plugins against its nodes. Developers are generally not security experts and also all of us make mistakes here-and-there, so it’s always nice to have a tool that can spot at least some of those security mistakes for us.
* *Coverage.py* — is a tool for measuring code coverage of *Python* programs. It gets triggered when we run our test suite with *Pytest* and generates coverage report from the test run. These reports can be in the form of terminal output, but also XML format which then can be consumed by CI tools.

With that out of the way, let’s go over what we have in setup.cfg. For *Flake8* we define exclusion patterns so that we don't lint code that we don't care about. Below that is an empty ignore section in case we need to ignore some rule globally. We also set max line length to 120, as keeping line length to 80 is in my opinion unreasonable with the size of today's screens. Final line sets *McCabe* complexity threshold to 10, if you are not familiar with *cyclomatic complexity* you can find out more [here](https://en.wikipedia.org/wiki/Cyclomatic_complexity).

Next up is *Bandit*, all we configure here is target directory, which is the name of our package. We do this so that we can avoid specifying targets on the command line.

After that follows *Coverage.py*. First, we enable *branch coverage*, which means that in places where a line in your program could jump to more than one next line, *Coverage.py* tracks which of those destination lines are actually visited. Next, we omit some files that shouldn’t or can’t be included in coverage measurement, like tests themselves or virtual environment files. We also exclude specific lines, e.g. lines that are labeled with pragma: no cover comment. Last *Coverage.py* config line tells the tool to store generated reports in reports directory. This directory is created automatically if it doesn't exist already.

The final tool we need to configure is *Pylint*, the configuration though, is *very* extensive, like more than 100 lines… So, I will leave this one out and point you the source [here](https://github.com/MartinHeinz/python-project-blueprint/blob/master/setup.cfg) as well as commented and explained pylintrc in *Pylint* repository [here](https://github.com/PyCQA/pylint/blob/master/pylintrc).

We went through all the tools in setup.cfg but there is one more that cannot be added to setup.cfg and that is *Pytest* - even though *Pytest* docs tell you that you can use setup.cfg, it's not exactly true... As per [this issue](https://github.com/pytest-dev/pytest/issues/3062#issuecomment-393523260), the option to use setup.cfg is being deprecated and there are some bugs like interpolation errors, that won't be fixed, therefore we will also need pytest.ini file for configuration of *Pytest*:

The first thing we do here is set a bunch of command line arguments — we enable colors in terminal output, then we enable coverage reporting for blueprint directory, after that we enable both generations of XML and stdout ( term) coverage reports. Final 2 arguments (-ra) tell *Pytest* to output short summary for non-passing tests.

On the next line, we have filterwarnings option which allows us to disable some annoying warnings in the output, for example, deprecation warnings coming out of some library which we have no control over.

Rest of the config sets up logging. First one just turns it on and other 3 configure level, format and datetime format. Easier than explaining the format config is just seeing the output itself, which is shown in the next section.

With all the configurations in pytest.ini, all we will need to do to run our test suite is run pytest, not even the package argument needed!

Last actual configuration file we have is requirement.txt, which contains a list of our dependencies. All you can find in this file is a list of *Python* packages, one per line with the *optional* version of the package. As noted, the package version is optional, but I strongly suggest you lock versions in requirements.txt to avoid situations, where you might download newer, *incompatible* package during build and deployment, and end-up breaking your application.

There are 2 remaining files which aren’t actually config files — our *Dockerfiles*, namely, dev.Dockerfile and prod.Dockerfile used for development and production images respectively. I will leave those out for time being as we will explore those in a separate article where we will talk about CI/CD and deployment. You can, however, check those files out already in *GitHub* repository here - <https://github.com/MartinHeinz/python-project-blueprint/blob/master/dev.Dockerfile>.

Actual Source Code

We have done quite a lot without even mentioning the source code of our application, but I think it’s time to look at those few lines of code that are in the project skeleton:

Only actual source code in this blueprint is this one class with a static method. This is really on needed so that we can run something, get some output and test it. This also works as entrypoint to the whole application. In a real project, you could use the run() method to initialize your application or web server.

So, how do we actually run this piece of code?

This short snippet in a specially named file \_\_main\_\_.py is what we need in our project so that we can run the whole package using python -m blueprint. The nice thing about this file and it's contents is that it will *only* be run with that command, therefore if we want to just import something from the source of this package without running the whole thing, then we can do so without triggering Blueprint.run().

There’s one more special file in our package and that’s the \_\_init\_\_.py file. Usually, you would leave it empty a use it only to tell *Python* that the directory is a package. Here, however, we will use it to export classes, variables and functions from our package.

Without this one line above you wouldn’t be able to call Blueprint.run() from outside of this package. This way we can avoid people using internal parts of our code that should not be exposed.

That’s all for the code of our package, but what about the tests? First, let’s look at the context.py

Normally when you use someone's package, then you import it like import blueprint or from blueprint import Blueprint, to imitate this in our tests and therefore make it as close as possible to real usage we use context.py file to import the package into our test context. We also insert our project root directory into system path. This is not actually necessary when running tests with pytest, but if you for example run context.py directly with python ./tests/context.py or possibly with unittest without including the sys.path.insert..., then you would get ModuleNotFoundError: No module named 'blueprint', so this one line is a little bit of *insurance policy*.

Now, let’s see the example test:

What we have here is just a single test that checks the standard output of Blueprint.run() using built-in *Pytest* fixture called capsys (capture system output). So, what happens when we run the test suite?

I trimmed a few lines from the output so that you can better see the relevant parts of it. What’s to note here? Well, our test passed! Other than that, we can see coverage report and we can also see that the report got written to coverage.xml as configured in pytest.ini. One more thing that we have here in the output is 2 log messages coming from conftest.py. What is that about?

You might have noticed that apart from capsys fixture, we also used example\_fixture in parameters of our small test. This fixture resides in conftest.py as should all custom fixtures we make:

As the name implies, this really is just an example fixture. All it does is log one message, then it lets the test run and finally, it logs one more message. The nice thing about conftest.py file is that it gets automatically discovered by *Pytest*, so you don’t even need to import it to your test files. If you want to find out more about it, then you can check out my previous post about *Pytest* [here](https://towardsdatascience.com/pytest-features-that-you-need-in-your-testing-life-31488dc7d9eb) or docs [here](https://docs.pytest.org/en/latest/fixture.html#conftest-py-sharing-fixture-functions).

One Command for Everything

It would be quite laborious if we were to run each of our tools separately and had to remember their arguments, even though they are always the same. Also, it would be equally annoying if later we decided to put all these tools into CI/CD (next article!), right? So, let’s simplify things with Makefile:

In this Makefile we have 4 targets. First of them - run runs our application using \_\_main\_\_.py we created in the root of our source folder. Next, test just runs pytest. It's that simple thanks to all the configs in pytest.ini. The longest target here - lint - runs all our linting tool. First, it runs pylint against all .py files in the project, including test files. After that it runs flake8 and finally bandit. For these 2 it runs only against sources in blueprint directory. If any of those tools find some problem with our code, it will exit with non-zero code, meaning the target will fail, which will be useful in CI/CD. Last target in this file is clean, which well... cleans our projects - it removes all the files generated by previously mentioned tools.

Conclusion

In this article we’ve built project skeleton, that’s ready to be used for any kind of *Python* project you might be working on or thinking about, so if you want to play with or dig a little deeper, then check out the source code which is available in my repository here: <https://github.com/MartinHeinz/python-project-blueprint>. Repo also includes information on how to set up your project using convenience script, plus some more docs. Feel free to leave feedback/suggestions in the form of issue or just star it if you like this kind of content. 🙂

In the future, we will look into adding CI/CD into the mix with *GitHub Actions* and *GitHub Package Registry*. We will also Dockerize our project and create both debuggable and optimized production ready Docker images and add some more code quality tooling using *CodeClimate* and *SonarCloud*.

Resources

* [Sample Python Module Repository](https://github.com/navdeep-G/samplemod)
* [Pytest Docs](https://docs.pytest.org/en/latest/contents.html)
* [Python Code Quality Authority](https://github.com/PyCQA)