**Setting up Python Projects: Part I**

Mastering the Art of Python Project Setup: A Step-by-Step Guide



Photo by [Yang Deng](https://unsplash.com/@palon?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

Whether you’re a seasoned developer or just getting started with 🐍 **Python**, it’s important to know how to build robust and maintainable projects. This tutorial will guide you through the process of setting up a Python project using some of the most popular and effective tools in the industry. You will learn how to use [GitHub](https://github.com/) and [GitHub Actions](https://github.com/features/actions) for version control and continuous integration, as well as other tools for testing, documentation, packaging and distribution. The tutorial is inspired by resources such as [Hypermodern Python](https://medium.com/@cjolowicz/hypermodern-python-d44485d9d769) and [Best Practices for a new Python project](https://mitelman.engineering/blog/python-best-practice/automating-python-best-practices-for-a-new-project/" \t "_blank). However, this is not the only way to do things and you might have different preferences or opinions. The tutorial is intended to be beginner-friendly but also cover some advanced topics. In each section, you will automate some tasks and add badges to your project to show your progress and achievements.

The repository for this series can be found at [github.com/johschmidt42/python-project-johannes](https://github.com/johschmidt42/python-project-johannes)

**Requirements**

* **OS**: Linux, Unix, macOS, Windows (WSL2 with e.g. Ubuntu 20.04 LTS)
* **Tools**: python3.10, bash, git, tree
* **Version Control System (VCS) Host**: [GitHub](https://github.com/)
* **Continuous Integration (CI) Tool**: [GitHub Actions](https://github.com/features/actions)

It is expected that you are familiar with the versioning control system (VCS) [git](https://git-scm.com/" \t "_blank). If not, here’s a refresher for you: [Introduction to Git](https://realpython.com/python-git-github-intro/" \t "_blank)

Commits will be based on [best practices for git commits](https://deepsource.io/blog/git-best-practices/" \t "_blank) & [Conventional commits](https://www.conventionalcommits.org/en/v1.0.0/" \t "_blank). There is the [conventional commit plugin](https://plugins.jetbrains.com/plugin/13389-conventional-commit" \t "_blank) for PyCharm or a [VSCode Extension](https://github.com/vivaxy/vscode-conventional-commits" \t "_blank) that help you to write commits in this format.

**Overview**

* **Part I (GitHub, IDE, Python environment, configuration, app)**
* [Part II (Formatting, Linting, Command management, CI)](https://johschmidt42.medium.com/setting-up-python-projects-part-ii-c4bd84b709d1)
* [Part III (Testing, CI)](https://johschmidt42.medium.com/setting-up-python-projects-part-iii-56aafde8ae0b)
* [Part IV (Documentation, CI/CD)](https://johschmidt42.medium.com/setting-up-python-projects-part-iv-82059eba4ca4)
* [Part V (Versioning & Releases, CI/CD)](https://johschmidt42.medium.com/setting-up-python-projects-part-v-206df3c1e3d3)
* [Part VI (Containerisation, Docker, CI/CD)](https://johschmidt42.medium.com/setting-up-python-projects-part-vi-cbdbf28eff53)

**Structure**

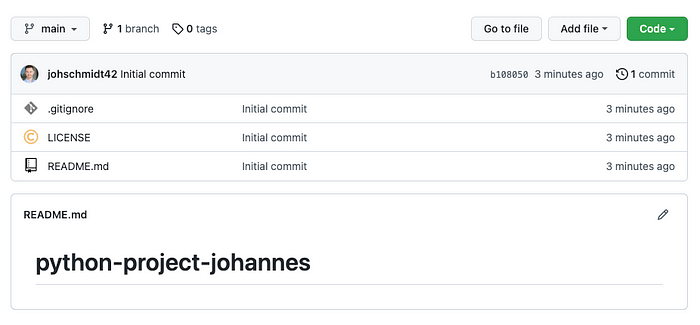
This part covers the following topics:

* Version Control System Host (*GitHub*)
* IDE (*PyCharm*, *VSCode*)
* Setting up a Python project (*src layout*)
* Setting up a Python environment (*Poetry*)
* Package configuration (*pyproject.toml*)
* Create an application (*fastAPI*)
* Bonus (*.editorconfig*)

**Version Control System Host**

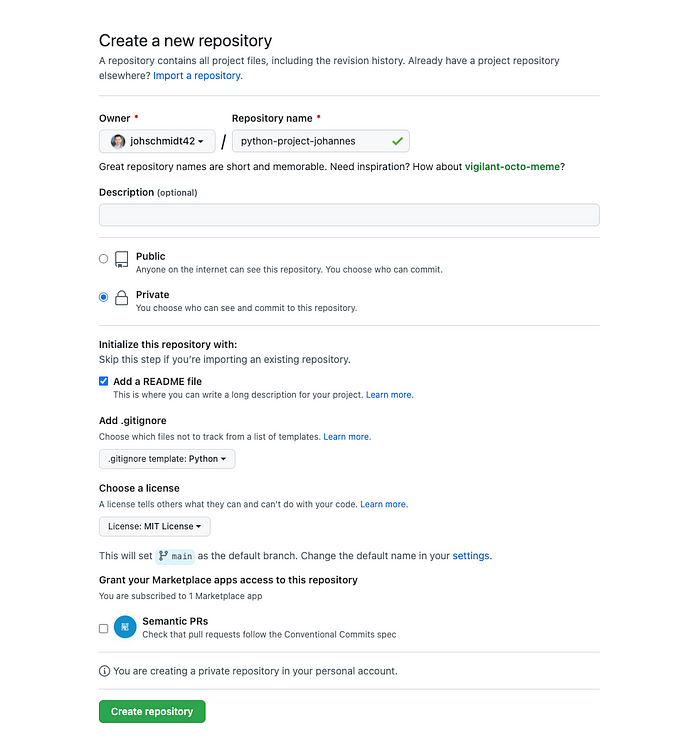
[GitHub](https://github.com/) is used to host the git repository in this series. Alternatives include for example: [GitLab](https://about.gitlab.com/" \t "_blank), [Bitbucket](https://bitbucket.org/" \t "_blank), [Azure DevOps](https://azure.microsoft.com/en-us/products/devops/) etc.

We will start off by creating a *private* repository on GitHub, so that we will have only these three files in our repo:



A new (“empty”) repository on GitHub (Image by author)

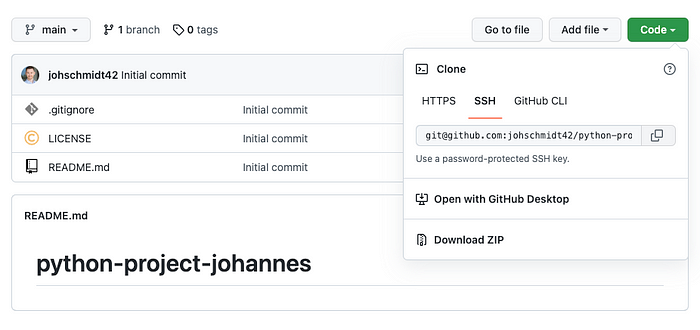
You can create a repository (*public* or *private*) on GitHub’s website for free if you log into your GitHub account:



Creating a new repository on GitHub (Image by author)

By choosing these settings, you will create three files in your repository: *README.md*, *.gitignore* and *LICENSE*. The *.gitignore* file is a default one for Python projects. It tells git which files or paths should be ignored and not tracked. I picked the [MIT license](https://opensource.org/license/mit/) because it allows anyone to use the code in this repository. If you are not sure which license to use, you can visit [choosealicense.com](https://choosealicense.com/) for some guidance. The *README.md* file only has a header with the name of the project for now: *python-project-johannes*

You can change the name to whatever you like. Next, we can clone our new repository with git using e.g. ssh in a bash terminal to get a local copy on our system:



Cloning a repository from GitHub with ssh (Image by author)

**> git clone git@github.com:johschmidt42/python-project-johannes.git**  
  
Cloning into 'python-project-johannes'...  
remote: Enumerating objects: 5, done.  
remote: Counting objects: 100% (5/5), done.  
remote: Compressing objects: 100% (4/4), done.  
Receiving objects: 100% (5/5), done.  
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0

If you have trouble cloning your repository with ssh, you might need to set up ssh keys on your system (or GitHub) or add the key to your *[ssh-agent](https://www.ssh.com/academy/ssh/agent" \t "_blank)*. You can find more information about this [here](https://docs.github.com/en/authentication/connecting-to-github-with-ssh" \t "_blank).

Once you clone the repository successfully, you can change the directory (cd) into the repo and check the content with ls or tree .

**> cd python-project-johannes**  
**> tree**  
.  
├── LICENSE  
└── README.md  
  
0 directories, 2 files

tree is a [command line tool](https://linux.die.net/man/1/tree" \t "_blank), that recursively lists the content of a directory. Please note that *.gitignore* and some other files are omitted by default but can be shown with the -a flag: tree -a

**IDE**

An integrated development environment (IDE) is a tool that helps you write code and manage a project. It usually offers features such as debugging, code completion, environment management, refactoring, version control and many more. These features can make your life easier, but they are not mandatory! However, I strongly suggest using an IDE for Python development.

The two most popular IDEs in the Python world are [PyCharm](https://www.jetbrains.com/pycharm/" \t "_blank) & [VisualStudio Code](https://code.visualstudio.com/" \t "_blank) right now. I personally use the PyCharm Community Edition, as it is my favourite one. Sometimes I use VSCode, and I like some of its features or extensions, but I always go back to PyCharm. You can use any IDE you want or none at all.

**Setting up a Python project**

There are many different [layouts](https://realpython.com/python-application-layouts/" \t "_blank) for a number of different application types, but the most popular one and recommended by the [Python Packaging Authority](https://packaging.python.org/tutorials/packaging-projects/) is the src layout. If you’re interested in why that is, I recommend to read up on the [advantages](https://py-pkgs.org/04-package-structure.html" \l "the-source-layout" \t "_blank) for python packaging.

We will create an example package, called example\_app that will follow the src layout and add a Python file app.py to this package.

.  
├── LICENSE  
├── README.md  
└── src  
 └── example\_app  
 ├── \_\_init\_\_.py  
 └── app.py  
  
2 directories, 4 files

**Setting up a Python environment**

If you want to work on Python software, you should *always* have a virtual environment. You do not want to risk breaking your main system environment. You want full control over versions of libraries that you installed. Virtual environments are disposable, while it is very hard and cumbersome to cleanup or update a main system environment.

There are many tools to create Python environments, some of which are:

* [venv](https://docs.python.org/3/library/venv.html)
* [pipx](https://github.com/pypa/pipx)
* [conda](https://anaconda.org/anaconda/conda) (preferably [miniconda](https://docs.conda.io/en/latest/miniconda.html" \t "_blank))
* [poetry](https://python-poetry.org/)

[Poetry](https://python-poetry.org/), for instance, handles everything: **environment management**, **dependency management**, **building**, **publishing**. It’s the new kid on the block in the Python ecosystem and I find a lot of value in this tool, as you will see throughout this series. That’s why I will use **Poetry** with **Python 3.10**. With its recent update to **1.2.0**, it now has some cool features such as **dependency groups** that were recently added ([Announcing Poetry 1.2.0](https://python-poetry.org/blog/announcing-poetry-1.2.0/" \t "_blank)) and that we will use. You can follow the instructions on how to install it on your OS. For Linux, macOS and Windows with WSL we can run:

**> curl -sSL https://install.python-poetry.org | python3 -**

To check if it has been installed, you can open a new shell and check the version of poetry with:

**> poetry --version**  
  
Poetry (version 1.2.0)

Once installed you can use the init command to initialise poetry in your project:

**> poetry init --name example\_app --no-interaction**

This will generate a simple *pyproject.toml* file in the current directory with some project information. I will explain this file later.

To create and activate a new environment, we can run:

**> poetry env use python3.10**  
  
Creating virtualenv example-app-PQrrHFSg-py3.10 in /Users/johannes/Library/Caches/pypoetry/virtualenvs  
Using virtualenv: /Users/johannes/Library/Caches/pypoetry/virtualenvs/example-app-PQrrHFSg-py3.10

Poetry uses [venv](https://docs.python.org/3/library/venv.html" \t "_blank) under the hood, for more information about managing environments in Poetry, check out: [Managing environments](https://python-poetry.org/docs/managing-environments/" \t "_blank).

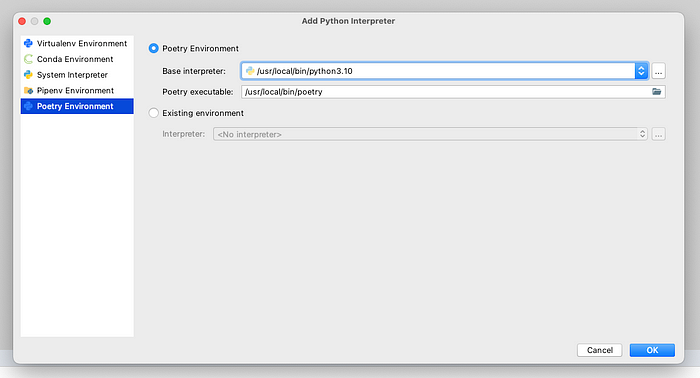
If you want to know where your virtual environment is, then you can run:

**> poetry env list --full-path**  
  
/Users/johannes/Library/Caches/pypoetry/virtualenvs/example-app-PQrrHFSg-py3.10 (Activated)

**Please note**: By default, Poetry only allows one venv per *pyproject.toml* file for the same Python interpreter.

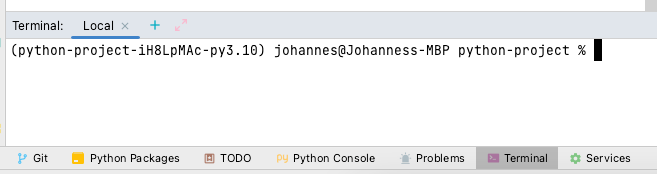
**PyCharm:**

In PyCharm, you can add a new poetry environment through the UI:



Creating a new Python environment in PyCharm (Image by author)

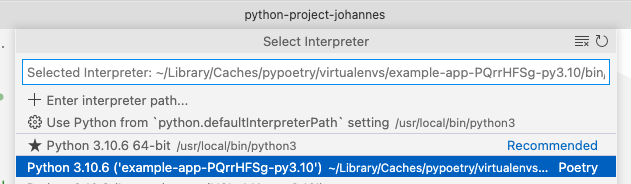
If we open a terminal in PyCharm, we see that this virtual environment is activated automatically. Neat!



Activated Python environment in PyCharm Terminal (Image by author)

**VSCode:**

In VSCode, we can add the interpreter through the Python: Select Interpreter command:



Creating a new Python environment in VSCode (Image by author)

If we open a terminal in VSCode, we also see that this virtual environment is activated automatically:



Activated Python environment in VSCode Terminal (Image by author)

**Package configuration**

Earlier when we created a virtual environment with Poetry (or venv), we initialised poetry in our project with the init command. This created the *pyproject.toml*. I haven’t explained this file yet but promised to do so. So here’s an explanation: It’s a [TOML](https://realpython.com/python-toml/) file that contains the entire package configuration. This package configuration file has been added to python in [PEP 517](https://peps.python.org/pep-0517/) and [518](https://peps.python.org/pep-0518/). By using Poetry, we automatically added some metadata to this file:

# pyproject.toml  
  
[tool.poetry]  
name = "example-app"  
version = "0.1.0"  
description = ""  
authors = ["Johannes Schmidt <johannes.schmidt.vik@gmail.com>"]  
readme = "README.md"  
packages = [{include = "example\_app"}]  
  
[tool.poetry.dependencies]  
python = "^3.10"  
  
[build-system]  
requires = ["poetry-core"]  
build-backend = "poetry.core.masonry.api"

It tells us:

* what backend to build the package: **build-system (**Instead of setup.py/setup.cfg or others, the backend to build the package is poetry**)**
* some metadata for the project: **tool.poetry** (the initial version of the package\*, description etc.)
* the dependencies for the app: **tool.poetry.dependencies**

**Please note**: The name in **[tool.poetry]** should be the same as the package. Otherwise, as it is in this case, we need to specify it with the packages field according to the [docs](https://python-poetry.org/docs/pyproject/#packages).

\*About versioning in Poetry:

*“While Poetry does not enforce any release convention, it does encourage the use of [semantic versioning](https://semver.org/" \t "_blank) within the scope of*[*PEP 440*](https://peps.python.org/pep-0440/#semantic-versioning)*.”*

Right now the version for our package is only in the *pyproject.toml* but it is standard to store it in the package’s *\_\_init\_\_.py*as per [PEP8](https://peps.python.org/pep-0008/):

# src/example\_app/\_\_init\_\_.py  
  
\_\_version\_\_ = "0.1.0"

Because there are only a few fields filled so far, I’m going to add some metadata manually:

# pyproject.toml  
  
[tool.poetry]  
...  
homepage = "https://github.com/johschmidt42/python-project-johannes"  
repository = "https://github.com/johschmidt42/python-project-johannes"  
license = "MIT"  
  
...

Currently, we don’t have any dependencies but Python in our repo, but we can create a lock file with

**> poetry lock**  
  
Updating dependencies  
Resolving dependencies... (0.1s)  
Writing lock file

It will create the *poetry.lock,*which is a lock file similar to JavaScript’s [package.json](https://blog.ezekielekunola.com/understanding-the-package.json-file" \t "_blank). When a developer *locks* a file, no one else can update that file until it is unlocked. It should be generally added to git.

In this lock file, we only have the required Python version right now. Let’s add a first required library for our *example\_app*:

**> poetry add httpx**  
  
Using version ^0.23.0 for httpx  
Updating dependencies  
Resolving dependencies... (0.8s)  
Writing lock file  
Package operations: 8 installs, 0 updates, 0 removals  
• Installing idna (3.4)  
• Installing sniffio (1.3.0)  
• Installing anyio (3.6.1)  
• Installing certifi (2022.9.24)  
• Installing h11 (0.12.0)  
• Installing httpcore (0.15.0)  
• Installing rfc3986 (1.5.0)  
• Installing httpx (0.23.0)

*HTTPX is a fully featured HTTP client for Python 3, which provides sync and async APIs, and support for both HTTP/1.1 and HTTP/2.*

We can check the installed dependencies with:

**> poetry show**  
  
anyio 3.6.1 High level compatibility layer for multiple asynchronous event loop implementations  
certifi 2022.9.24 Python package for providing Mozilla's CA Bundle.  
h11 0.12.0 A pure-Python, bring-your-own-I/O implementation of HTTP/1.1  
httpcore 0.15.0 A minimal low-level HTTP client.  
httpx 0.23.0 The next generation HTTP client.  
idna 3.4 Internationalized Domain Names in Applications (IDNA)  
rfc3986 1.5.0 Validating URI References per RFC 3986  
sniffio 1.3.0 Sniff out which async library your code is running under

At this point, our directory tree should resemble this:

.  
├── LICENSE  
├── README.md  
├── poetry.lock  
├── pyproject.toml  
└── src  
 └── example\_app  
 ├── \_\_init\_\_.py  
 └── app.py  
  
2 directories, 6 files

**Create an application**

We don’t want to add a dependency (*httpx*) to our requirements if we don’t use it in our application. So let’s write some code that uses the httpx library.

For this I will create a small [REST API](https://realpython.com/api-integration-in-python/) with [fastAPI](https://fastapi.tiangolo.com/" \t "_blank). If you are not familiar with this library, don’t worry you don’t need to understand everything to follow along. But I suggest you give it a try!

**> poetry add fastapi**

This will install [pydantic](https://pydantic-docs.helpmanual.io/" \t "_blank), [starlette](https://www.starlette.io/" \t "_blank) and [typing-extensions](https://github.com/python/typing_extensions" \t "_blank).

[Starlette](https://www.starlette.io/) is the power horse under the hood:

*Starlette is a lightweight*[*ASGI*](https://asgi.readthedocs.io/en/latest/)*framework/toolkit, which is ideal for building async web services in Python.*

To run our *fastAPI* application (e.g. locally), we need a server implementation:

**> poetry add "uvicorn[standard]"**

This will install uvicorn with “Cython-based” dependencies (where possible) and other “optional extras”. You can find more details [here](https://www.uvicorn.org/" \t "_blank). Cython does not ring a bell to you? Check out this [tutorial](https://realpython.com/cpython-source-code-guide/) if you’re interested.

Ok let us add some code to our *app.py*.

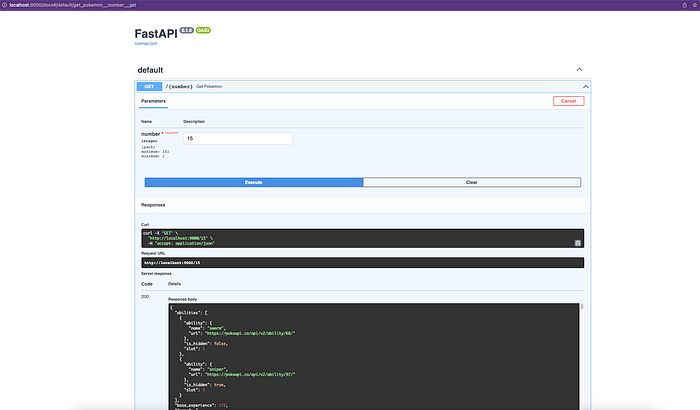
In essence, we created the *fastAPI* application and added a **GET** endpoint that receives a number between 1 and 151. This endpoint sends a HTTP request to the *pokeapi.co* domain with *httpx* to retrieve information about a Pokemon by number!

The if \_\_name\_\_ == “\_\_main\_\_": section allows us to run the application with *uvicorn*locally*.*We can run the app with:

**> python src/example\_app/app.py**  
  
INFO: Will watch for changes in these directories: ['/Users/johannes/workspace/python-project-johannes']  
INFO: Uvicorn running on http://127.0.0.1:9000 (Press CTRL+C to quit)  
INFO: Started reloader process [24844] using WatchFiles  
INFO: Started server process [24846]  
INFO: Waiting for application startup.  
INFO: Application startup complete.

This will launch the application on port 9000 and reload it automatically when files change.

In the browser we can open [*http://localhost:9000/docs*](http://localhost:9000/docs) to open the interactive documentation that fastAPI generated for us. [OpenAPI](https://swagger.io/specification/" \t "_blank) conform.



Interactive REST API documentation with Swagger (OpenAPI) (Image by author)

And we can directly call the endpoint by entering a number between 1 and 151 (click on Try it out ) to see in stdout:

INFO: 127.0.0.1:60689 - "GET /15 HTTP/1.1" 200 OK

It’s **[beedrill](https://pokemondb.net/pokedex/beedrill" \t "_blank)**!

That’s it for this tutorial. We created a repository on **GitHub**, cloned it, created a Python environment with **Poetry** and created a simple **fastAPI** application in **src** **layout** that we showed in action!

This part laid the basis of what comes in the next parts: **Linting**, **Testing**, **Documentation**, **Badges**, **Releases**, **Docker**, **CI/CD** and much more!