How to build python library and upload it in PyPi

We know that a module is a file with some Python code, and a package is a directory for sub packages and modules. But the line between a package and a Python library is quite blurred.

A Python library is a reusable chunk of code that you may want to include in your programs/ projects.

Compared to languages like C++ or C, a Python libraries do not pertain to any specific context in Python. Here, a ‘library’ loosely describes a collection of core modules.

Essentially, then, a library is a collection of modules. A package is a library that can be installed using a package manager like rubygems or npm.

Creating the package files

we will now add files that are used to prepare the project for distribution. When we’re done, the project structure will look like this:

package/

├── LICENSE

├── pyproject.toml

├── README.md

├── setup.cfg

├── src/

   └── quickcal/

   ├── \_\_init\_\_.py

Creating pyproject.toml

pyproject.toml tells build tools (like [pip](https://packaging.python.org/key_projects/#pip) and [build](https://packaging.python.org/key_projects/#build)) what is required to build your project. I have used [setuptools](https://packaging.python.org/key_projects/#setuptools), so open pyproject.toml and enter the following content:

**[build-system]**

requires = [

"setuptools>=42",

"wheel"

]

build-backend = "setuptools.build\_meta"

build-system.requires gives a list of packages that are needed to build your package. Listing something here will *only* make it available during the build, not after it is installed.

build-system.build-backend is the name of Python object that will be used to perform the build.

## Configuring metadata

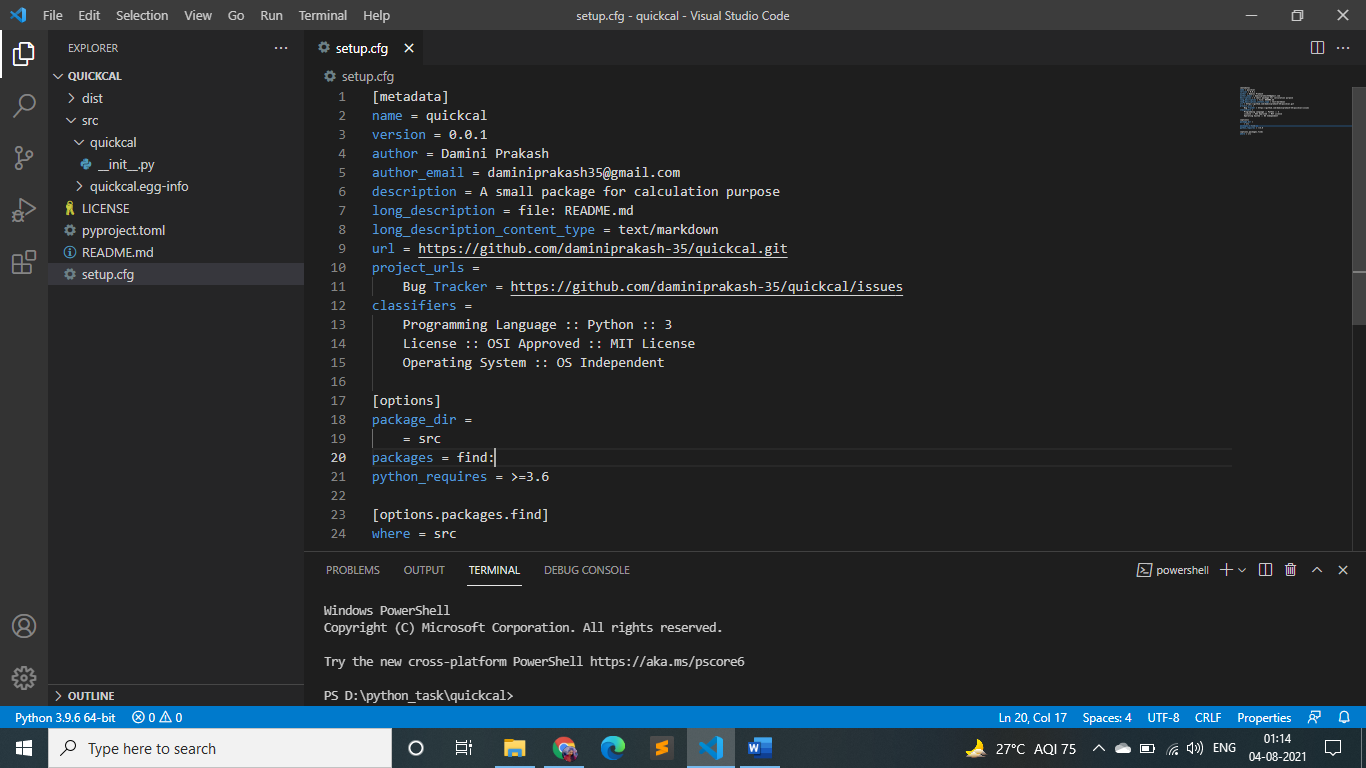
There are two types of metadata: static and dynamic.

* Static metadata (setup.cfg): guaranteed to be the same every time. This is simpler, easier to read, and avoids many common errors, like encoding errors.
* Dynamic metadata (setup.py): possibly non-deterministic. Any items that are dynamic or determined at install-time, as well as extension modules or extensions to setuptools, need to go into setup.py.

Static metadata (setup.cfg) should be preferred. Dynamic metadata (setup.py) should be used only as an escape hatch when absolutely necessary. setup.py used to be required, but can be omitted with newer versions of setuptools and pip.

setup.cfg is the configuration file for [setuptools](https://packaging.python.org/key_projects/" \l "setuptools). It tells setuptools about our package (such as the name and version) as well as which code files to include. Eventually much of this configuration may be able to move to pyproject.toml.

Open setup.cfg and enter the following content. Change the name to include our username; this ensures that we have a unique package name and that our package doesn’t conflict with packages uploaded by other people.



There are a [variety of metadata and options](https://setuptools.readthedocs.io/en/latest/userguide/declarative_config.html) supported here. This is in [configparser](https://docs.python.org/3/library/configparser.html" \o "(in Python v3.9)) format; do not place quotes around values. This example package uses a relatively minimal set of metadata:

* name is the distribution name of your package. This can be any name as long as it only contains letters, numbers, \_ , and -. It also must not already be taken on pypi.org. **Be sure to update this with your username,** as this ensures you won’t try to upload a package with the same name as one which already exists.
* version is the package version. You can use file: or attr: directives to read from a file or package attribute.
* author and author\_email are used to identify the author of the package.
* description is a short, one-sentence summary of the package.
* long\_description is a detailed description of the package. This is shown on the package detail page on the Python Package Index. In this case, the long description is loaded from README.md (which is a common pattern) using the file: directive.
* long\_description\_content\_type tells the index what type of markup is used for the long description. In this case, it’s Markdown.
* url is the URL for the homepage of the project. For many projects, this will just be a link to GitHub, GitLab, Bitbucket, or similar code hosting service.
* project\_urls lets you list any number of extra links to show on PyPI. Generally this could be to documentation, issue trackers, etc.
* classifiers gives the index and [pip](https://packaging.python.org/key_projects/#pip) some additional metadata about your package. In this case, the package is only compatible with Python 3, is licensed under the MIT license, and is OS-independent. You should always include at least which version(s) of Python your package works on, which license your package is available under, and which operating systems your package will work on.

In the options category, we have controls for setuptools itself:

* package\_dir is a mapping of package names and directories. An empty package name represents the “root package” — the directory in the project that contains all Python source files for the package — so in this case the src directory is designated the root package.
* packages is a list of all Python [import packages](https://packaging.python.org/glossary/#term-Import-Package) that should be included in the [distribution package](https://packaging.python.org/glossary/#term-Distribution-Package). Instead of listing each package manually, we can use the find: directive to automatically discover all packages and subpackages and options.packages.find to specify the package\_dir to use. In this case, the list of packages will be quickcal as that’s the only package present.
* python\_requires gives the versions of Python supported by your project. Installers like [pip](https://packaging.python.org/key_projects/#pip) will look back through older versions of packages until it finds one that has a matching Python version.

## Creating README.md

Open README.md and enter the content which we like to.

**# Example Package**

This is a simple example package. You can use

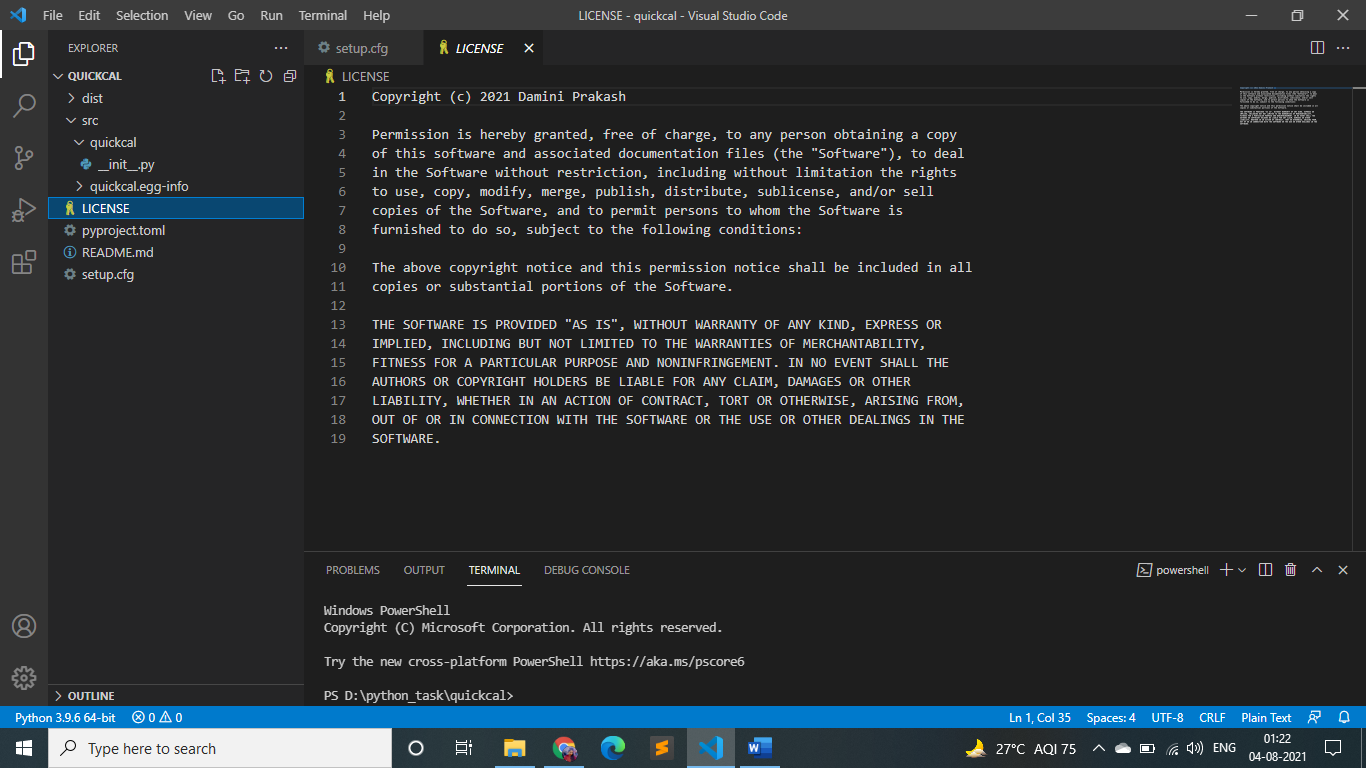
[**Github-flavored Markdown**](https://guides.github.com/features/mastering-markdown/)

to write your content.

Because our configuration loads README.md to provide a long\_description, README.md must be included along with our code when we [generate a source distribution](https://packaging.python.org/tutorials/packaging-projects/#generating-archives). Newer versions of [setuptools](https://packaging.python.org/key_projects/" \l "setuptools) will do this automatically.

## Creating a LICENSE

It’s important for every package uploaded to the Python Package Index to include a license. This tells users who install your package the terms under which they can use your package. For help picking a license, see <https://choosealicense.com/>. Once you have chosen a license, open LICENSE and enter the license text. For example, as I have chosen the MIT license:



## Generating distribution archives

The next step is to generate [distribution packages](https://packaging.python.org/glossary/#term-Distribution-Package) for the package. These are archives that are uploaded to the Python Package Index and can be installed by [pip](https://packaging.python.org/key_projects/#pip).

Make sure you have the latest version of PyPA’s [build](https://packaging.python.org/key_projects/#build) installed:

py -m pip install --upgrade build

Now run this command from the same directory where pyproject.toml is located:

py -m build

This command should output a lot of text and once completed should generate two files in the dist directory:

dist/

example\_package\_YOUR\_USERNAME\_HERE-0.0.1-py3-none-any.whl

example\_package\_YOUR\_USERNAME\_HERE-0.0.1.tar.gz

The tar.gz file is a [source archive](https://packaging.python.org/glossary/#term-Source-Archive) whereas the .whl file is a [built distribution](https://packaging.python.org/glossary/#term-Built-Distribution). Newer [pip](https://packaging.python.org/key_projects/#pip) versions preferentially install built distributions, but will fall back to source archives if needed. You should always upload a source archive and provide built archives for the platforms your project is compatible with. In this case, our quickcal package is compatible with Python on any platform so only one built distribution is needed.

## Uploading the distribution archives

Finally, it’s time to upload your package to the Python Package Index!

The first thing you’ll need to do is register an account on TestPyPI, which is a separate instance of the package index intended for testing and experimentation. It’s great for things like this tutorial where we don’t necessarily want to upload to the real index. To register an account, go to <https://test.pypi.org/account/register/> and complete the steps on that page. You will also need to verify your email address before you’re able to upload any packages. For more details, see [Using TestPyPI](https://packaging.python.org/guides/using-testpypi/).

To securely upload your project, you’ll need a PyPI [API token](https://test.pypi.org/help/#apitoken). Create one at <https://test.pypi.org/manage/account/#api-tokens>, setting the “Scope” to “Entire account”. **Don’t close the page until you have copied and saved the token — you won’t see that token again.**

Now that you are registered, you can use [twine](https://packaging.python.org/key_projects/#twine) to upload the distribution packages. You’ll need to install Twine:

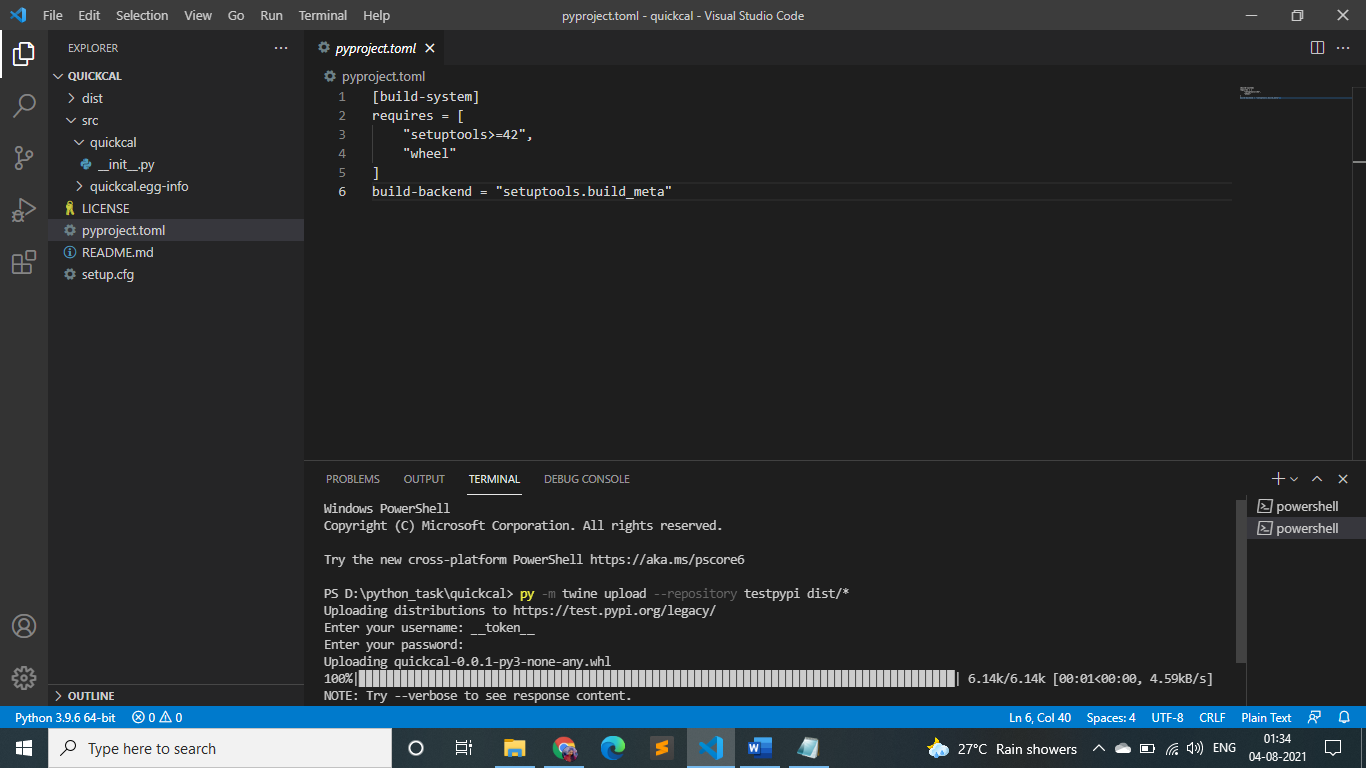
py -m pip install --upgrade twine

Once installed, run Twine to upload all of the archives under dist:

py -m twine upload --repository testpypi dist/\*

You will be prompted for a username and password. For the username, use \_\_token\_\_. For the password, use the token value, including the pypi- prefix.

After the command completes, you should see output



## Upload to Pypi

The Python Standard Library comprises of sophisticated and robust capabilities for working with larger packages. We will find modules for working with sockets and with files and file paths.

Though there might be great packages that Python comes with, there are more exciting and fantastic projects outside the standard library which are mostly called the [Python Packaging Index](https://pypi.org/) (PyPI). It is nothing but a repository of software for the Python programming language.

The PyPI package is considered as an important property for Python being a powerful language.

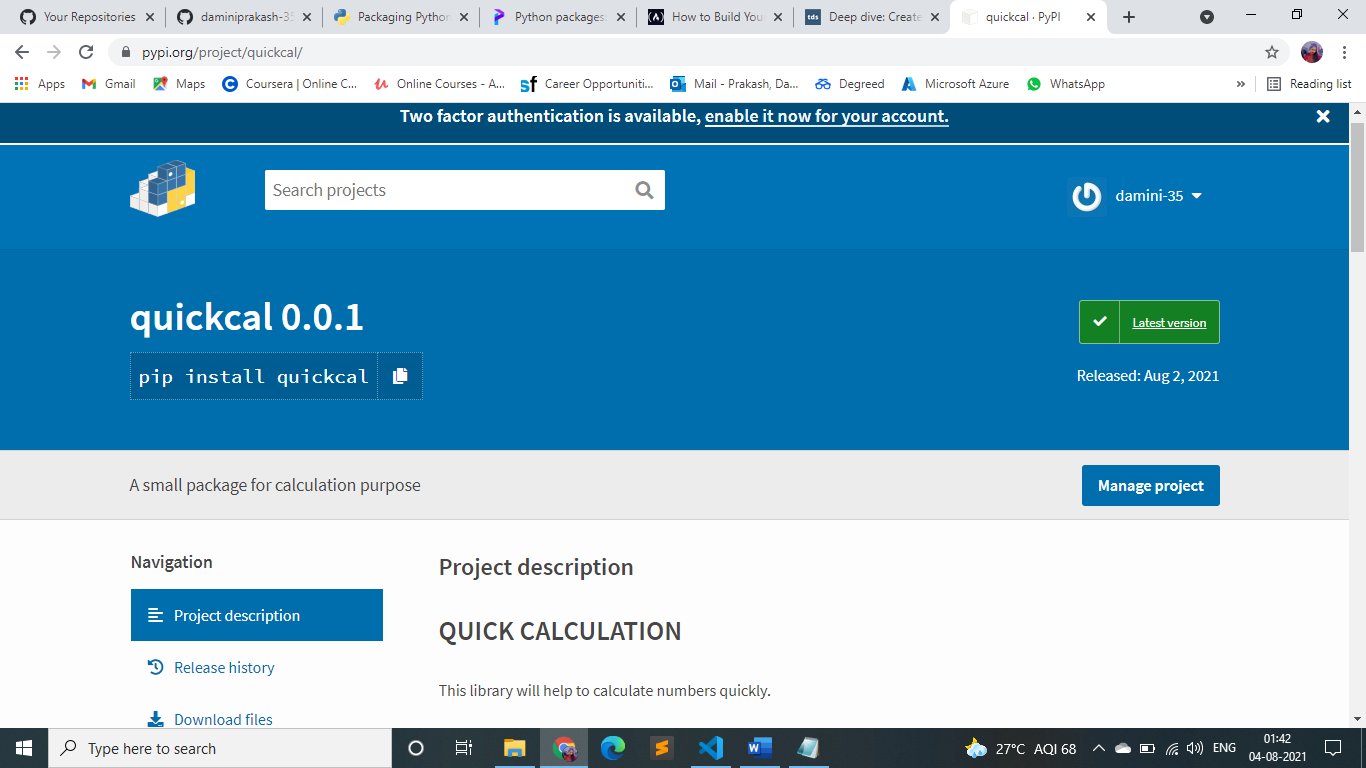
"PyPI" should be pronounced like "pie pea eye", specifically with the "PI" pronounced as individual letters, but rather as a single sound. This minimizes confusion with the [PyPy](http://www.pypy.org/" \t "_blank) project, which is a popular alternative implementation of the Python language.

The Python Package Index, abbreviated as PyPI is also known as the Cheese Shop. It is the official third-party [software repository](https://en.wikipedia.org/wiki/Software_repository) for Python, just like [CPAN](https://en.wikipedia.org/wiki/CPAN) is the repository for  [Perl](https://en.wikipedia.org/wiki/Perl).  Some package managers such as [pip](https://en.wikipedia.org/wiki/Pip_(Python)), use PyPI as the default source for packages and their dependencies. More than 113,000 Python packages can be accessed through PyPI.

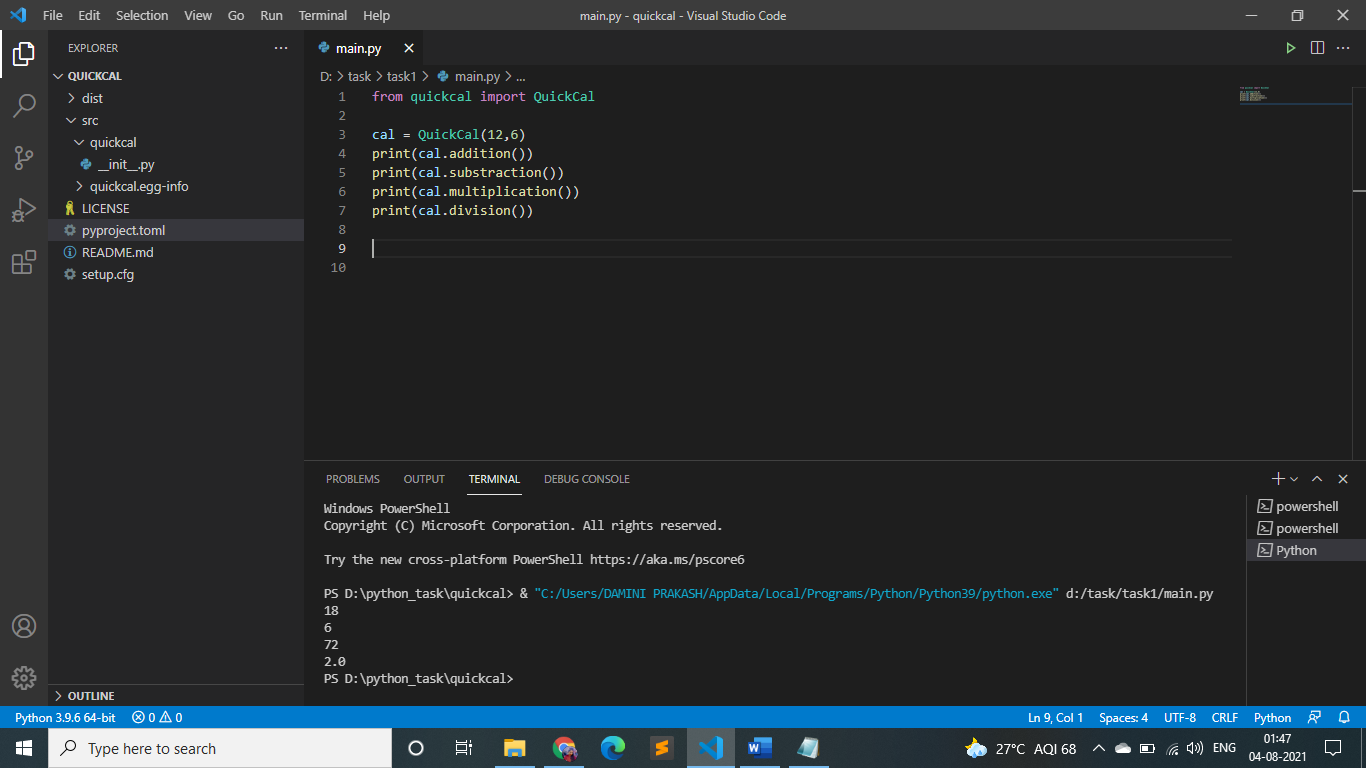
when we are satisfied with the result,we can publish it on the real Pypi.

twine upload dist/\*

Type the command above, connect to Pypi and our library is published. Anyone can now download and install it with “pip install quickcal’’.



Now we can use this library for further simple calculation as shown below:-



References

* <https://packaging.python.org/tutorials/packaging-projects/>
* <https://towardsdatascience.com/deep-dive-create-and-publish-your-first-python-library-f7f618719e14>
* <https://www.freecodecamp.org/news/build-your-first-python-package/>