**PACKAGING IN PYTHON**

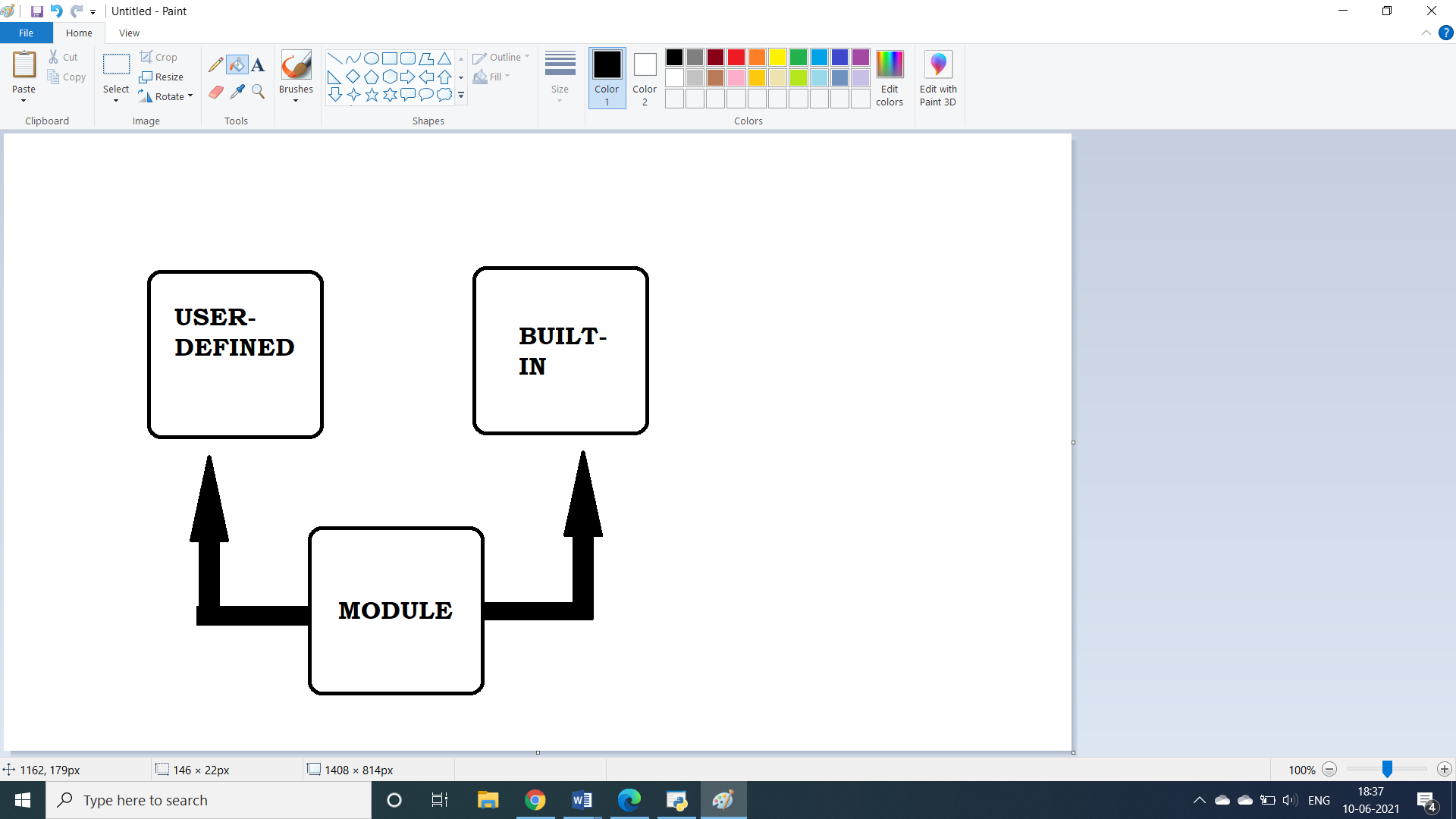
*Blog by Sandhya Reddy,* [*MSRF*](https://www.madrasresearch.org/) *intern*

**INTRODUCTION**:

Before knowing what a package is, it is important to understand what a module is in the first place. So, a module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

# **WHAT ARE PYTHON MODULES?**

A **Python module** is a Python file containing a set of functions and variables to be used in an application. The variables can be of any type (arrays, dictionaries, objects, etc.)



**Figure-1**

Modules can be either:

**1. Built in**

**2. User-defined.**

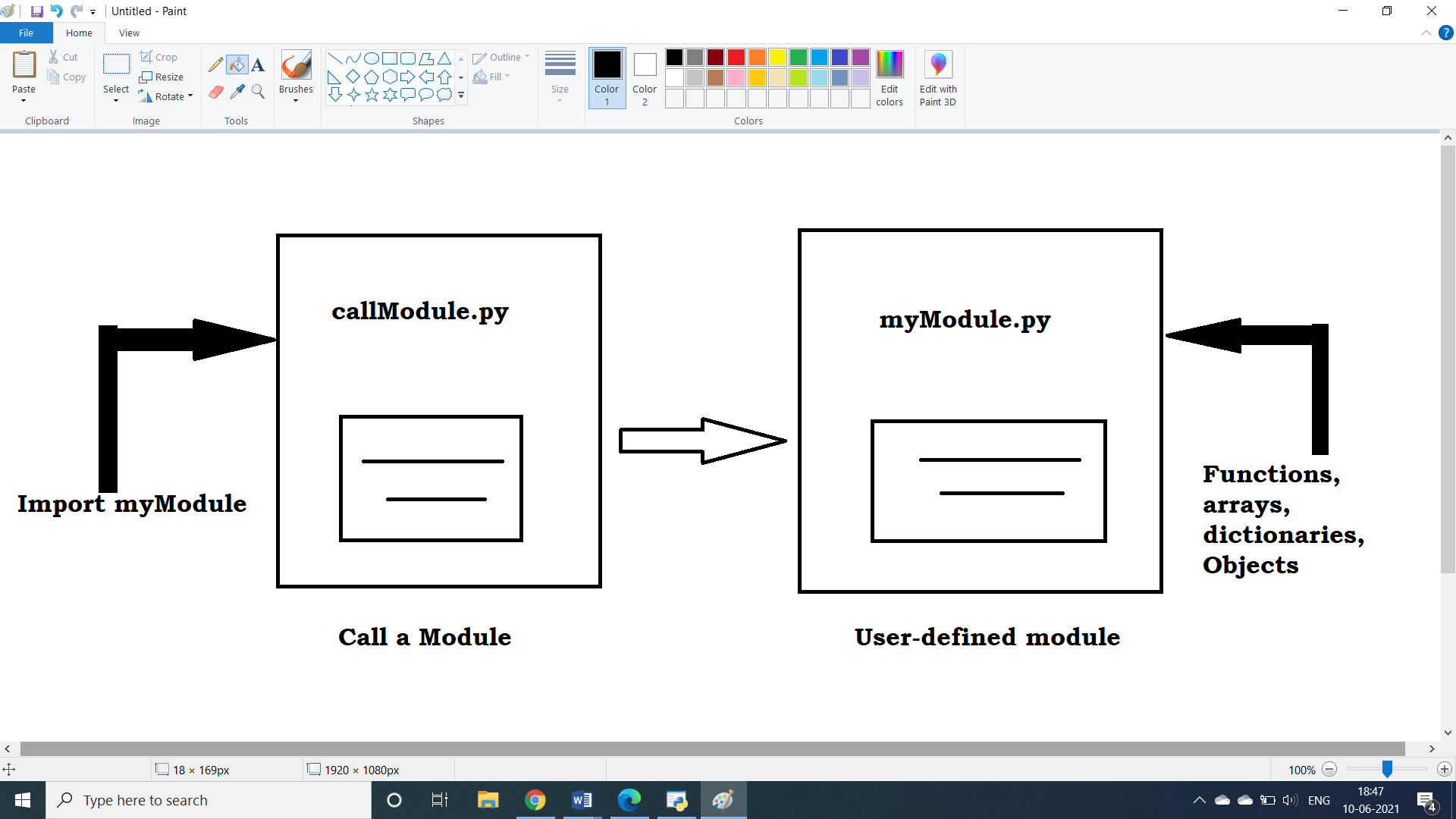
## 1) User-defined Modules

#### **Create a Module**

To create a module, create a Python file with a .py extension.

#### **Call a Module**

Modules created with a .py extension can be used in another Python source file, using the import statement.

 **Figure-2**

Here’s an example of a simple module, myModule.py:

#myModule.py

def myFunction( parameter ):  #define a function in myModule.py

   print "Course : ", parameter

    return

To call this Python module myModule.py, create another Python file callModule.py file and use the import statement.

import myModule.py:

   myModule.myFunction(“Python Programming”)

When the Python interpreter encounters an import statement, it imports the module if it is present in the **search path**. A search path is a list of directories that the interpreter searches for importing a module.

When the above code is executed, the following output is produced:

#callModule.py

import myModule

myModule.myFunction("Python Programming")

#myModule.py

def myFunction( parameter ):  #define a function in myModule.py

  print ("Course : ", parameter)

  return

## 2) Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

#### **Call a built-in Module**

To call a built-in Module and use the function of that module write:

import moduleName #call a module

moduleName.function()#use module function

import math

print("The value of cosine is", math.cos(3))

print("The value of sine is", math.sin(3))

print("The value of tangent is", math.tan(3))

print("The value of pi is", math.pi)

## Benefits of modules in Python

There are a couple of key benefits of creating and using a module in Python:

#### **Structured Code**

* Code is logically organized by being grouped into one Python file which makes development easier and less error-prone.
* Code is easier to understand and use.

#### **Reusability**

Functionality defined in a single module can be easily reused by other parts of the application. This eliminates the need to recreate duplicate code.

**WHAT IS A PACKAGE IN PYTHON?**

* A package is a collection of Python modules, i.e., a package is a directory of Python modules containing an additional \_\_init\_\_.py file.
* This means that the distinction between module and package is just at the file system level.
* In short, a package is a collection of Python modules: while a module is a single Python file, a package is a directory of Python modules containing an additional \_\_init\_\_.py file, to distinguish a package from a directory that just happens to contain a bunch of Python scripts.

**WHAT EXACTLY IS PACKAGING?**

* Packaging in python is creating, using and deploying packages. Any operation which has something directly to do with packages is packaging.
* Python’s native packaging is mostly built for distributing reusable code, called libraries, between developers. You can piggyback **tools**, or basic applications for developers, on top of Python’s library packaging, using technologies like [setuptools entry\_points](https://setuptools.readthedocs.io/en/latest/userguide/entry_point.html).

Libraries are building blocks, not complete applications. For distributing applications, there’s a whole new world of technologies out there.

## [PACKAGING PYTHON LIBRARIES AND TOOLS](https://packaging.python.org/overview/#id5)

A few of the tools Python’s ecosystem which provides for distributing Python code to developers are:

* PyPI
* Setup.py
* Wheel files

### [**PYTHON BINARY DISTRIBUTIONS**](https://packaging.python.org/overview/#id8)

So much of Python’s practical power comes from its ability to integrate with the software ecosystem, in particular libraries written in C, C++, Fortran, Rust, and other languages.

**WHEEL:**

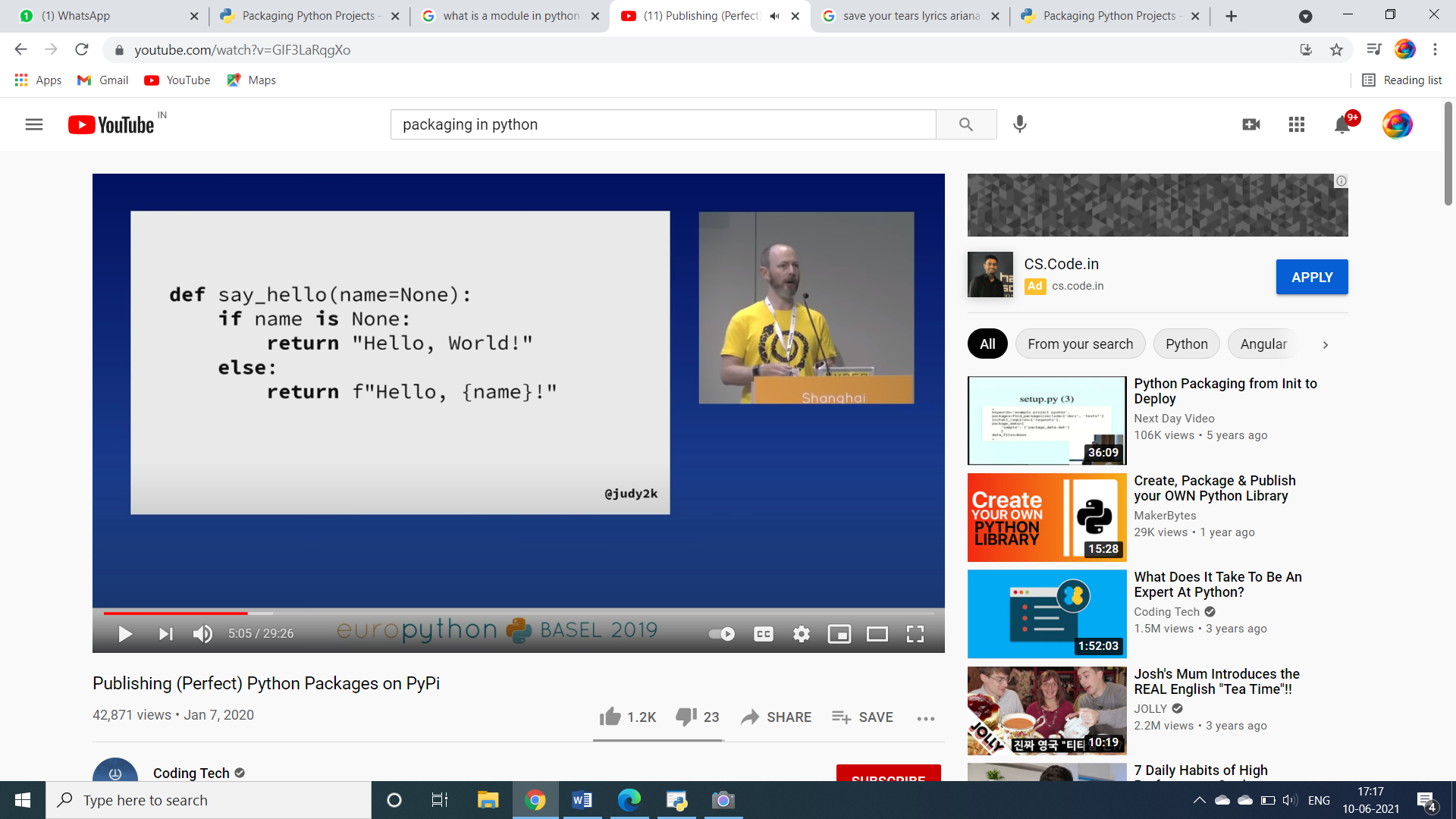
* Not all developers have the right tools or experiences to build these components written in these compiled languages.
* So Python created the [Wheel](https://packaging.python.org/glossary/#term-Wheel), a package format designed to ship libraries with compiled artifacts.
* In fact, Python’s package installer, pip, always prefers wheels because installation is always faster, so even pure-Python packages work better with wheels.

Binary distributions are best when they come with source distributions to match. Even if you don’t upload wheels of your code for every operating system, by uploading the sdist, you’re enabling users of other platforms to still build it for themselves. Default to publishing both sdist and wheel archives together, unless you’re creating artifacts for a very specific use case where you know the recipient only needs one or the other.

Python and PyPI make it easy to upload both wheels and sdists together.

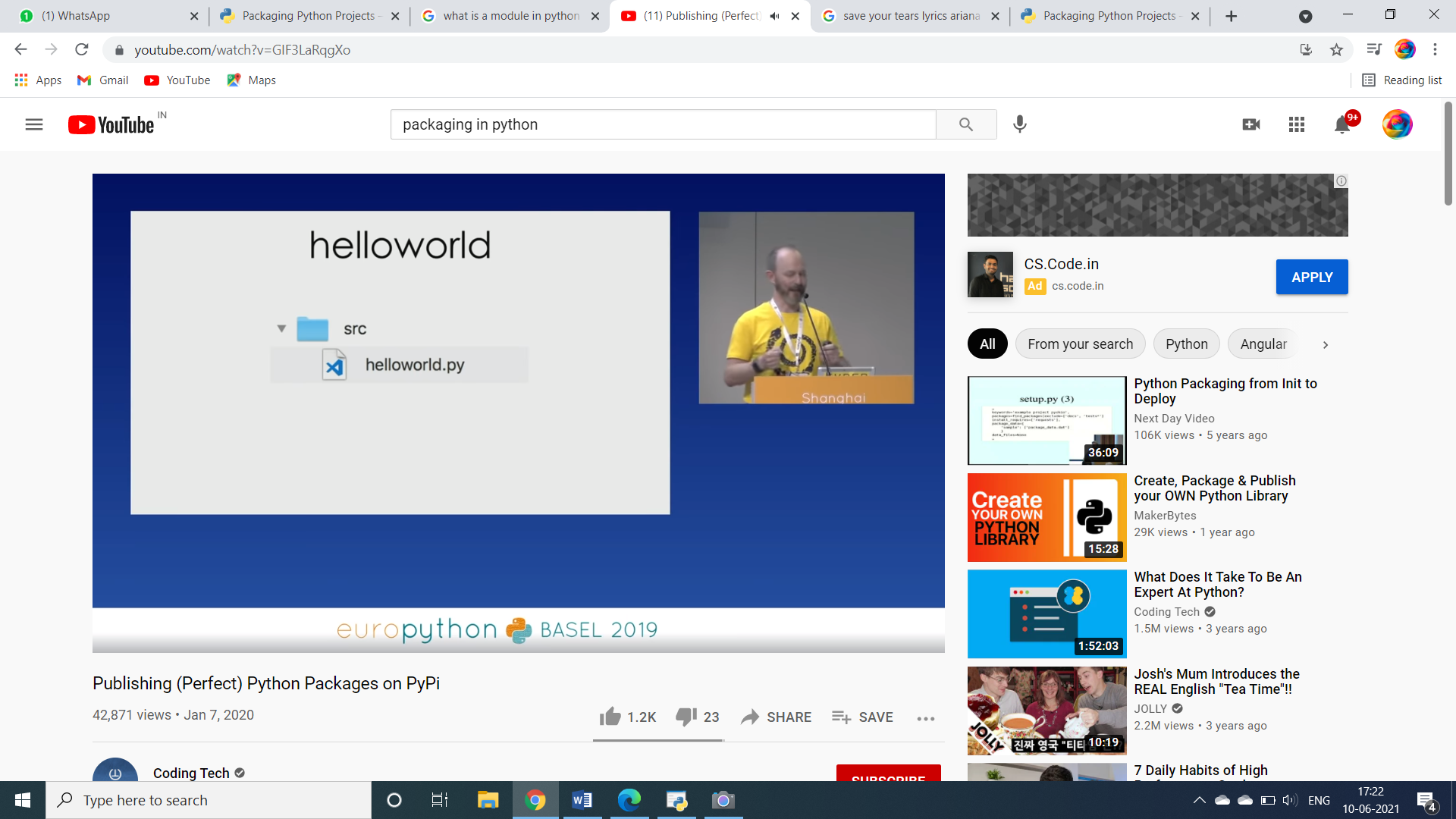
**LOCALLY MAKING AND IMPORTING A PACKAGE:**

For example, lets create a package named “helloword” which just prints “Hello, World” if name isn’t mentioned else “Hello, name”, pretty simple and straight forward:



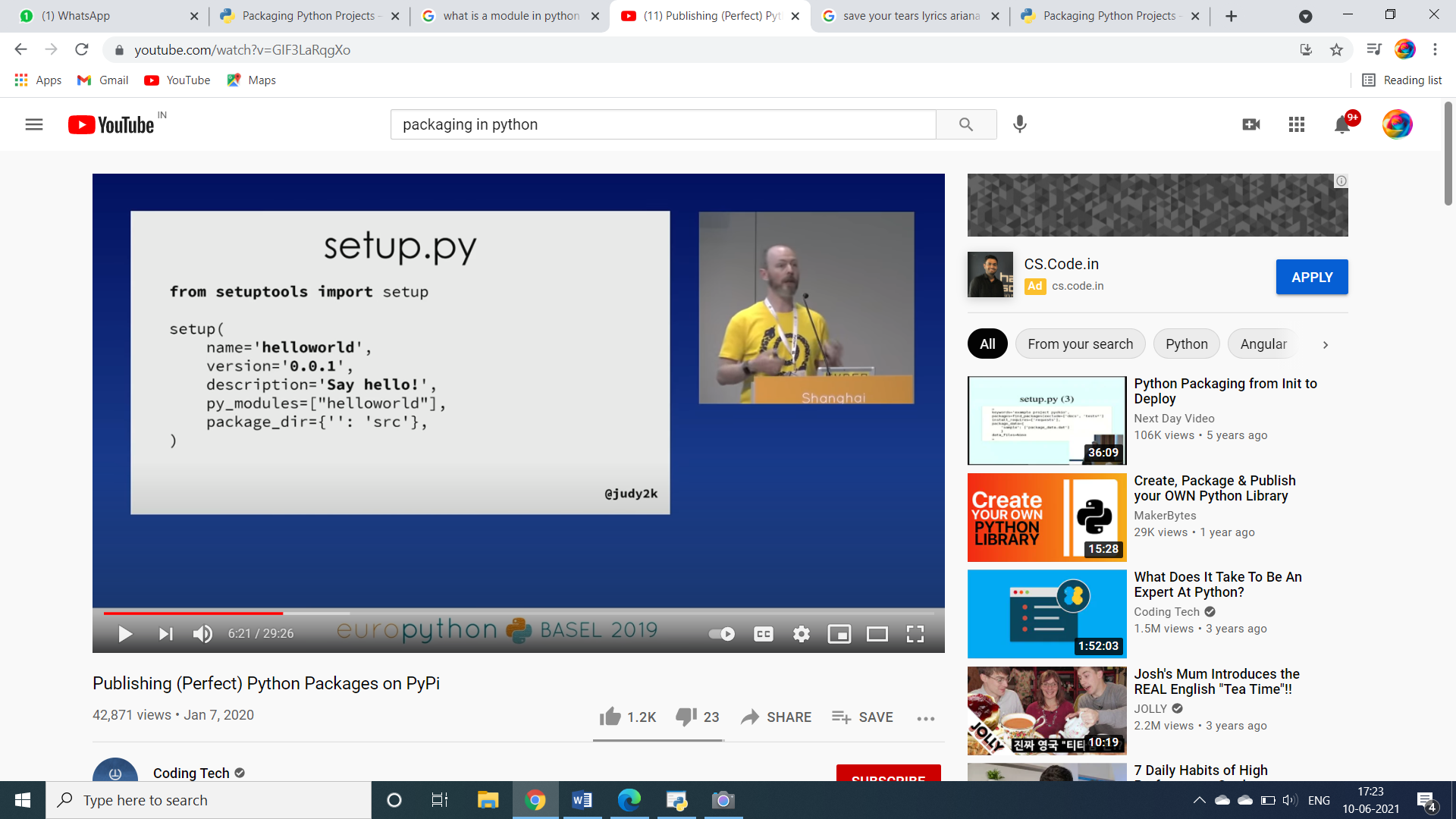
**Figure-3**

Save the code as “helloworld.py” and save the file in a folder called “src”.



**Figure-4**

Create a “setup.py” file and enter the following code:



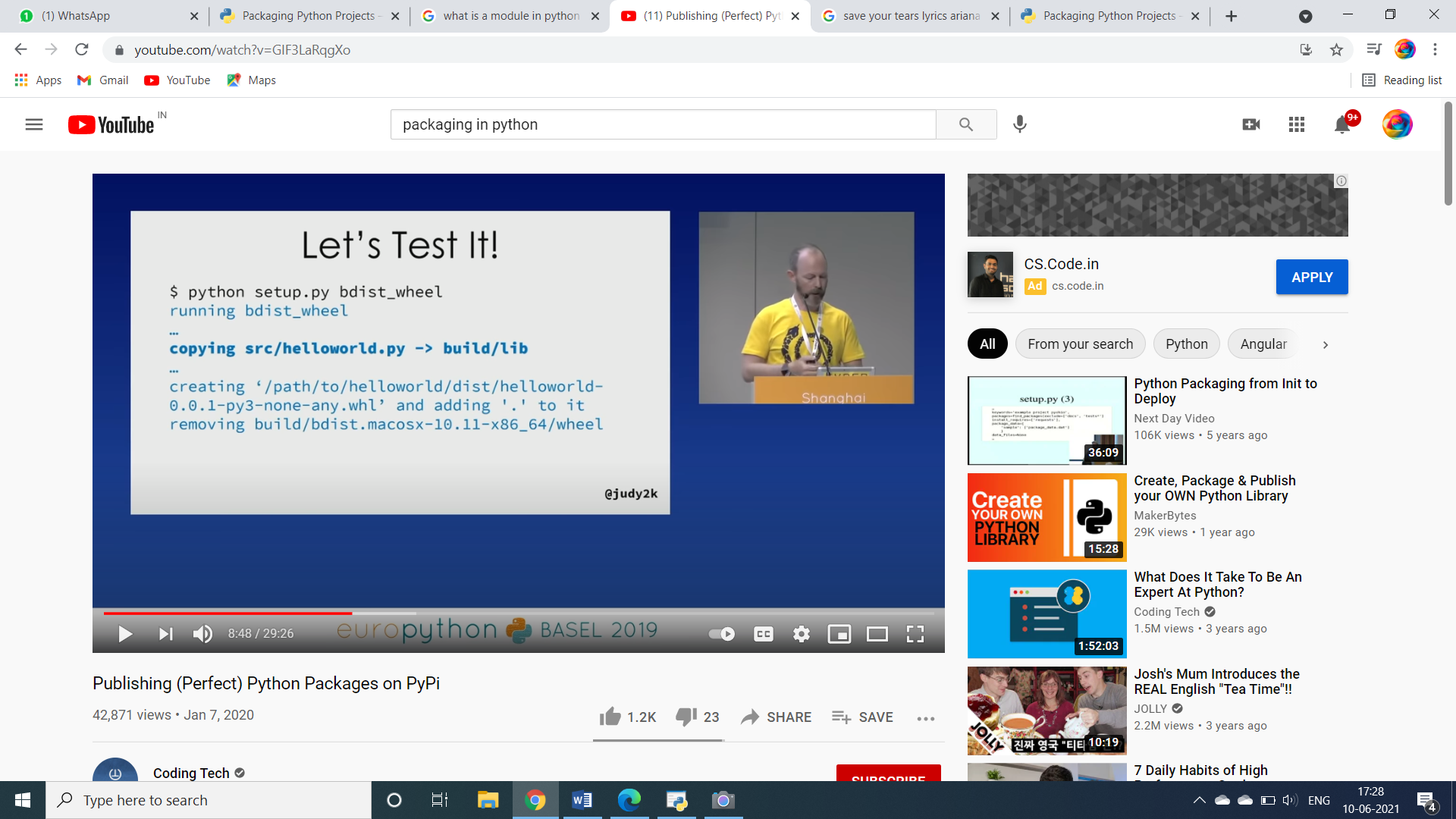
**Figure-5**

Setup function is basically a configuration which tells how to store the package. The version number could be “0.0.x”, for starter, x=1, is the recommended number to create a stable package locally.

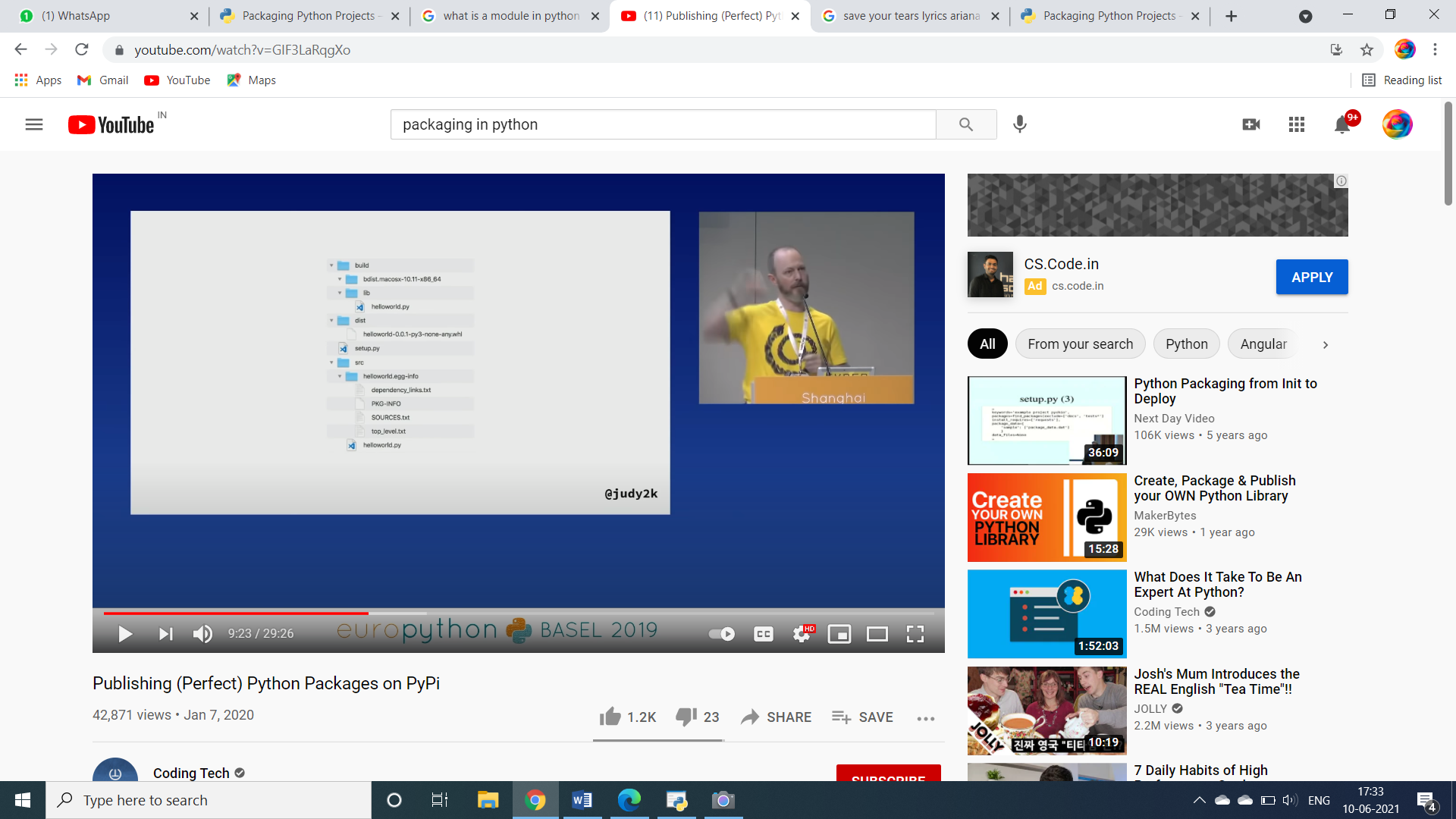
To test the package run the following code:

Python setup.py bdist\_wheel

It basically, creates a wheel file to builf PYPI. It copies the src folder along with the helloword.py file to the lib directory of python.



It automatically, builds the following file structure:

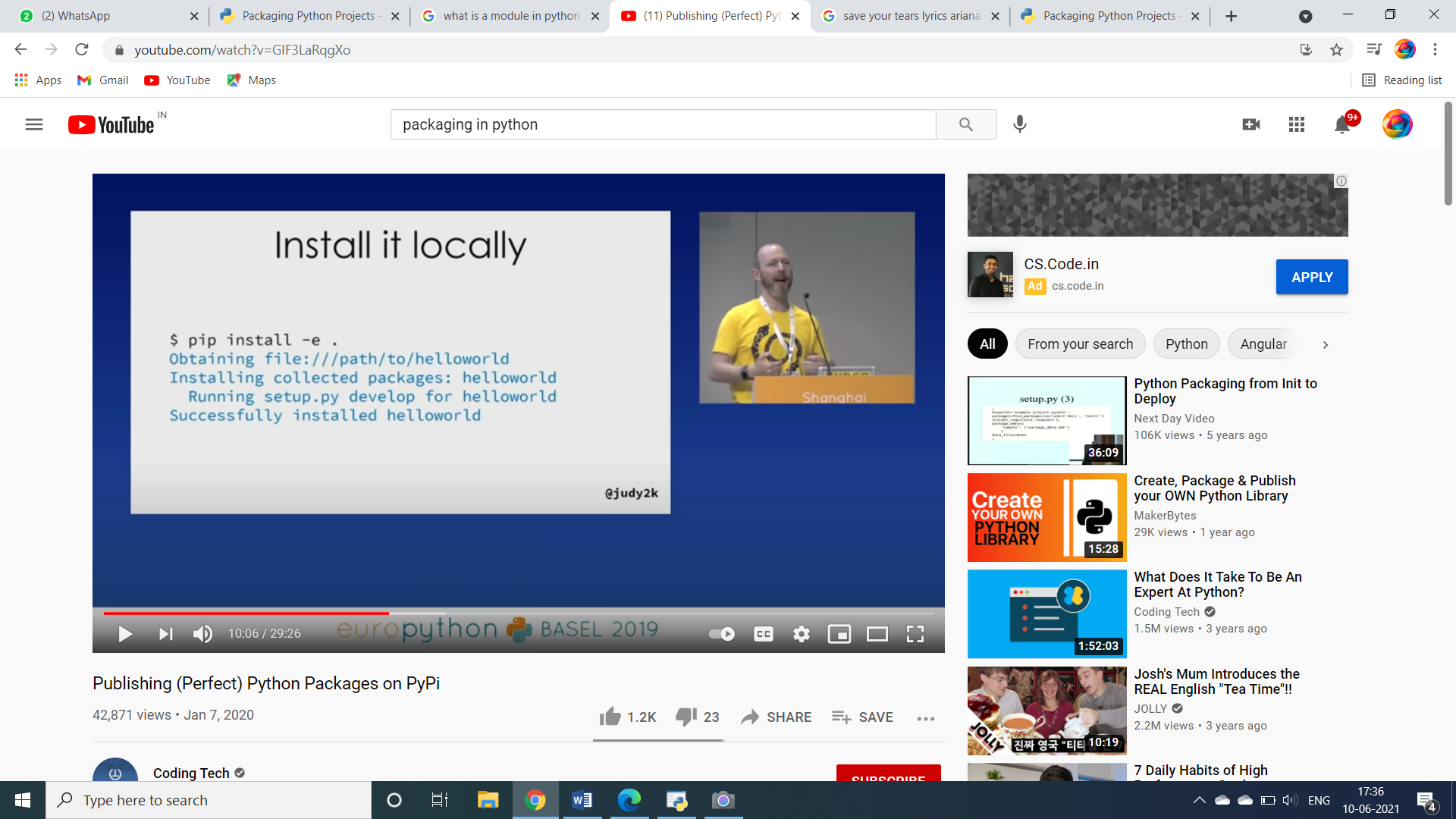


**Figure-6**

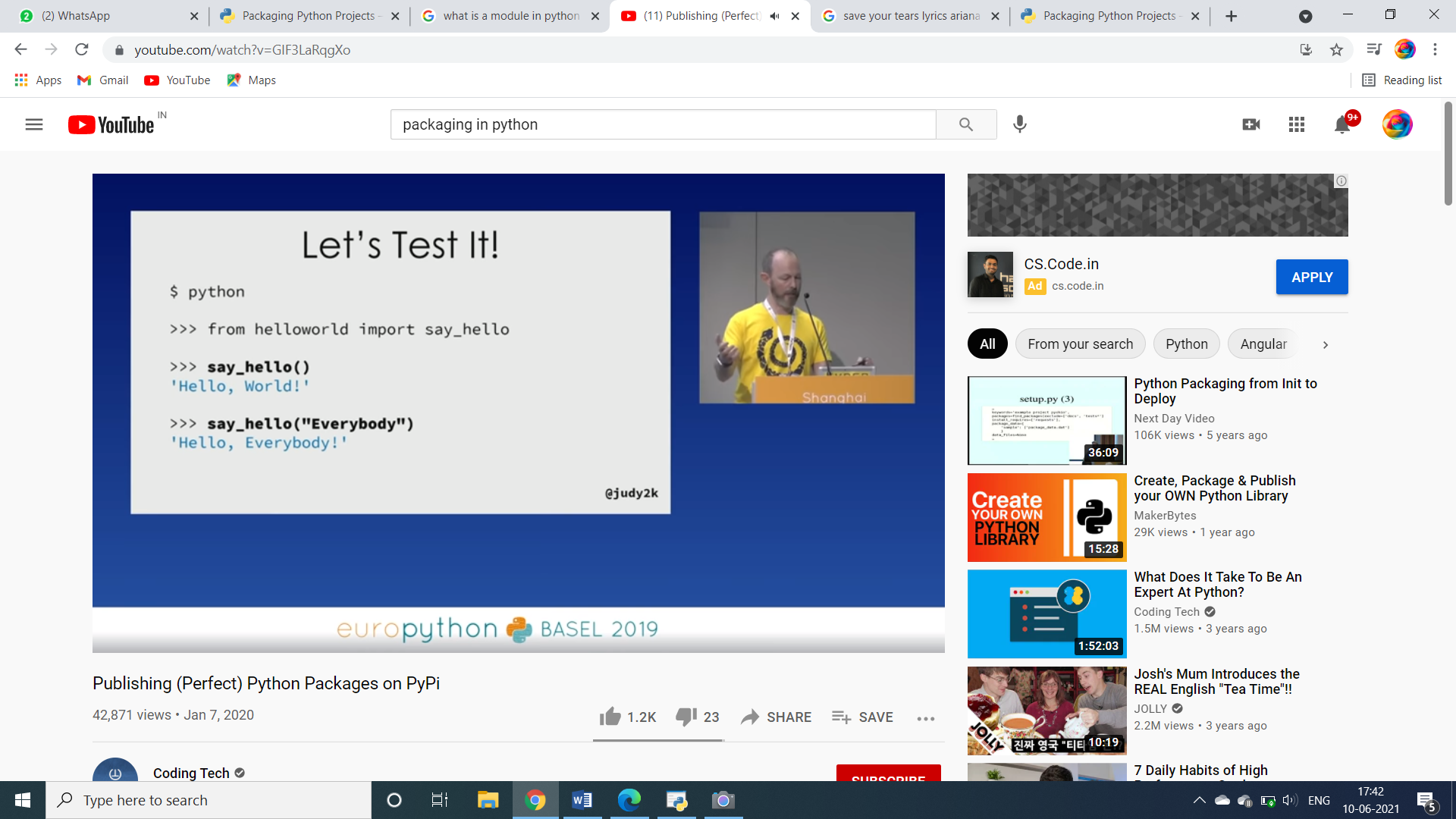
To install it locally, run the following code:

Pip install –e .

It is an essential command to build python packages. Usually, while installing a package, what happens is the file is transferred to python distributions in the system. But, we need to link the code we need to work on instead of copying it, this job is done by “-e “in the command. The “.” at the end means it installed the package in the current directory.



Let’s Test it:



**HOW TO PACKAGE A SIMPLE PYTHON PROJECT AND DEPLOY IT:**

**This can be done by the following 3 steps:**

1. Adding the necessary files and structure to create the package.
2. Building the package
3. Uploading it to the Python Package Index.

Adding the necessary files and structure to create the package:

Unix/macOS

python3 -m pip install --upgrade pip

Windows

py -m pip install --upgrade pip

## Create a simple project

A simple project named “example\_pkg” is created.

Create the following file structure locally:

* packaging\_tutorial/
* └── src/
* └── example\_pkg/
* └── \_\_init\_\_.py
* src/example\_pkg/\_\_init\_\_.py is required to import the directory as a package, and should be empty.
* Once the structure is created, run all of the commands within the packaging\_tutorial directory.

## Create the package files

Add files that are used to prepare the project for distribution. When it is done, the project structure should look like this:

* packaging\_tutorial/
* ├── LICENSE
* ├── pyproject.toml
* ├── README.md
* ├── setup.cfg
* ├── src/
* │   └── example\_pkg/
* │       └── \_\_init\_\_.py
* └── tests/

## Create a test directory

tests/ is a placeholder for test files. Leave it empty for now.

## Create 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. Enter the following content because setuptools is being used:

[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. If you were to use a different build system, such as [flit](https://packaging.python.org/key_projects/#flit) or [poetry](https://packaging.python.org/key_projects/#poetry), those would go here, and the configuration details would be completely different than the [setuptools](https://packaging.python.org/key_projects/#setuptools) configuration described below.

## 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 (static)

setup.cfg is the configuration file for [setuptools](https://packaging.python.org/key_projects/#setuptools). It tells setuptools about your 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 your username; this ensures that you have a unique package name and that your package doesn’t conflict with packages uploaded by other people following this tutorial.

[metadata]

name = example-pkg-YOUR-USERNAME-HERE

version = 0.0.1

author = Example Author

author\_email = author@example.com

description = A small example package

long\_description = file: README.md

long\_description\_content\_type = text/markdown

url = https://github.com/pypa/sampleproject

project\_urls =

    Bug Tracker = https://github.com/pypa/sampleproject/issues

classifiers =

    Programming Language :: Python :: 3

    License :: OSI Approved :: MIT License

    Operating System :: OS Independent

[options]

package\_dir =

    = src

packages = find:

python\_requires = >=3.6

[options.packages.find]

where = src

## Create a README.md

Open README.md and enter the following content. You can customize this if you’d like.

# 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 your code when you [generate a source distribution](https://packaging.python.org/tutorials/packaging-projects/#generating-archives). Newer versions of [setuptools](https://packaging.python.org/key_projects/#setuptools) will do this automatically.

## Create 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, if you had chosen the MIT license:

Copyright (c) 2018 The Python Packaging Authority

Permission is hereby granted, free of charge, to any person obtaining a copy

of this software and associated documentation files (the "Software"), to deal

in the Software without restriction, including without limitation the rights

to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

SOFTWARE.

## Include other files

* The files listed above will be included automatically in your [source distribution](https://packaging.python.org/glossary/#term-Source-Distribution-or-sdist).
* If you want to control what goes in this explicitly, see [Including files in source distributions with MANIFEST.in](https://packaging.python.org/guides/using-manifest-in/#using-manifest-in).
* The final [built distribution](https://packaging.python.org/glossary/#term-Built-Distribution) will have the Python files in the discovered or listed Python packages.
* If you want to control what goes here, such as to add data files, see [Including Data Files](https://setuptools.readthedocs.io/en/latest/userguide/datafiles.html) from the [setuptools docs](https://setuptools.readthedocs.io/en/latest/index.html).

## Generate 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:

 Unix/macOS

python3 -m pip install --upgrade build

Windows

py -m pip install --upgrade build

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

Unix/macOS

python3 -m build

Windows

py -m build

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

dist/

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

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

## 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:

Windows

py -m pip install --upgrade twine

Unix/macOS

python3 -m pip install --upgrade twine

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

Unix/macOS

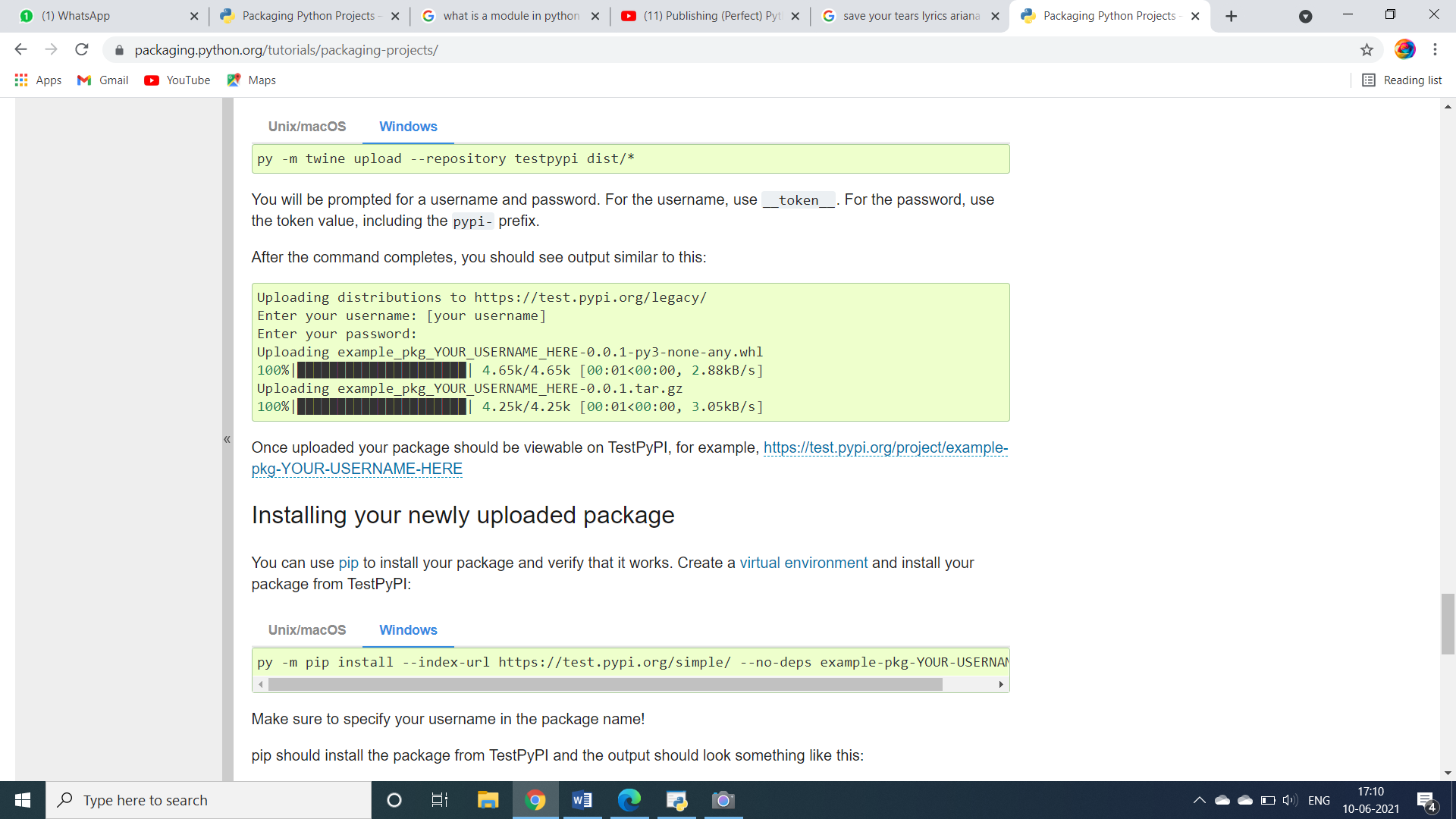
python3 -m twine upload --repository testpypi dist/\*

Windows

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 similar to this:



Once uploaded your package should be viewable on TestPyPI, for example, <https://test.pypi.org/project/example-pkg-YOUR-USERNAME-HERE>.

## Installing the newly uploaded package

You can use [pip](https://packaging.python.org/key_projects/#pip) to install your package and verify that it works. Create a [virtual environment](https://packaging.python.org/tutorials/installing-packages/#creating-and-using-virtual-environments) and install your package from TestPyPI:

Unix/macOS

python3 -m pip install --index-url https://test.pypi.org/simple/ --no-deps example-pkg-YOUR-USERNAME-HERE

WindowsOS

py -m pip install --index-url https://test.pypi.org/simple/ --no-deps example-pkg-YOUR-USERNAME-HERE

Make sure to specify your username in the package name!

pip should install the package from TestPyPI and the output should look something like this:

Collecting example-pkg-YOUR-USERNAME-HERE

Downloading https://test-files.pythonhosted.org/packages/.../example-pkg-YOUR-USERNAME-HERE-0.0.1-py3-none-any.whl

Installing collected packages: example-pkg-YOUR-USERNAME-HERE

Successfully installed example-pkg-YOUR-USERNAME-HERE-0.0.1

You can test that it was installed correctly by importing the package. Make sure you’re still in your virtual environment, then run Python:

Unix/macOS

python3

Windows

py

and import the package:

>>> import example\_pkg

**CONGRATULATIONS, YOU’VE PACKAGED AND DISTRIBUTED A PYTHON PROJECT!**

**On Conclusion, in this blog the following topics were covered:**

1. What is a module in python?
2. What is a package in python?
3. What exactly is packaging?
4. Various packaging libraries and tools.
5. How to create a package and install it locally?
6. How to make a package and deploy it?

For complete code, checkout the following github link:

**REFERENCES**:

* <https://www.youtube.com/watch?v=GIF3LaRqgXo>
* <https://packaging.python.org/tutorials/packaging-projects/>
* <https://www.educative.io/edpresso/what-are-python-modules>