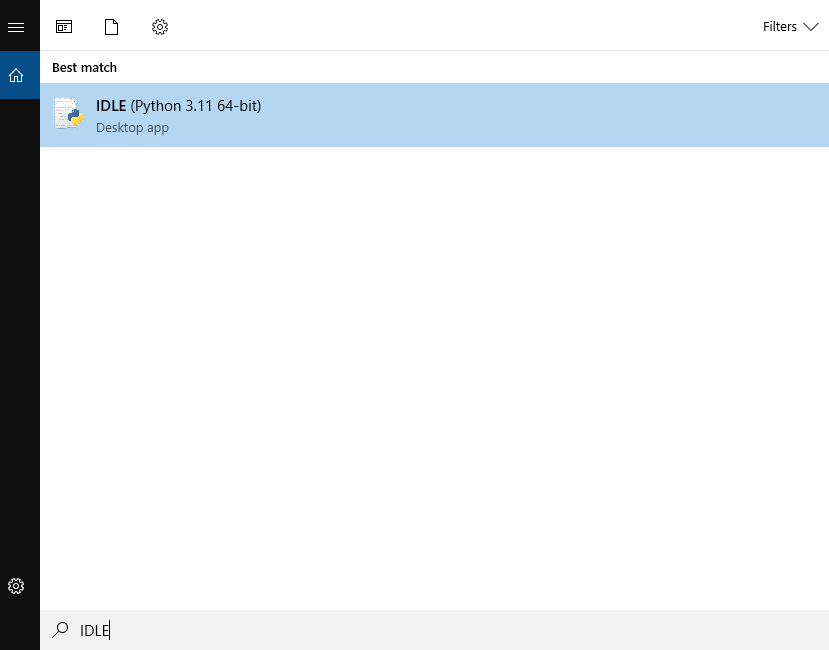
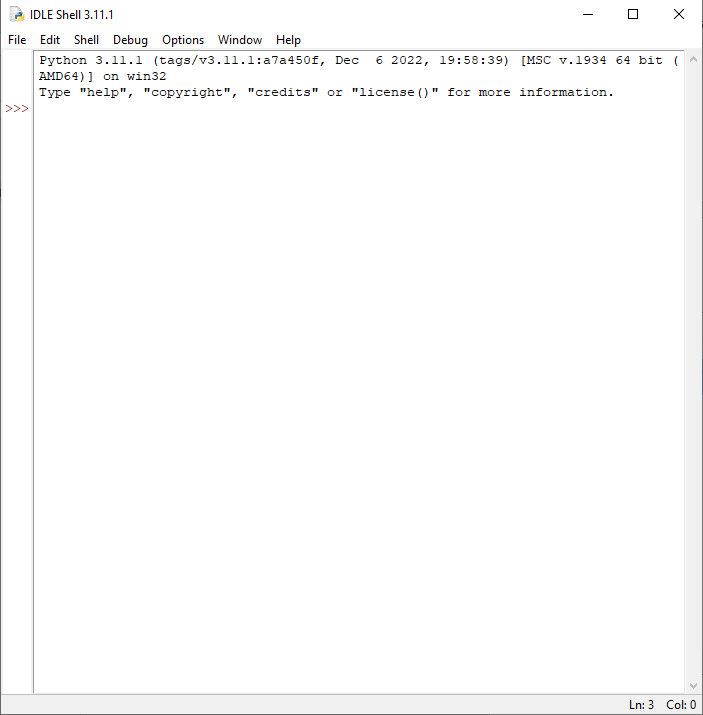
The thing that you actually write code in is called an Integrated Development Environment (IDE). The most basic one is called IDLE for Python.

OK, now I have python! How do I code?

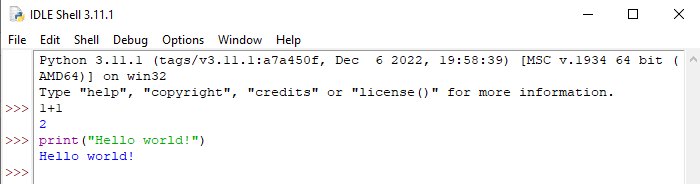
Well, one of the easiest ways is to use IDLE. If you have Python already installed on your computer, a simple search should bring it up. It is built into Python.



When you open it, it should look like this.

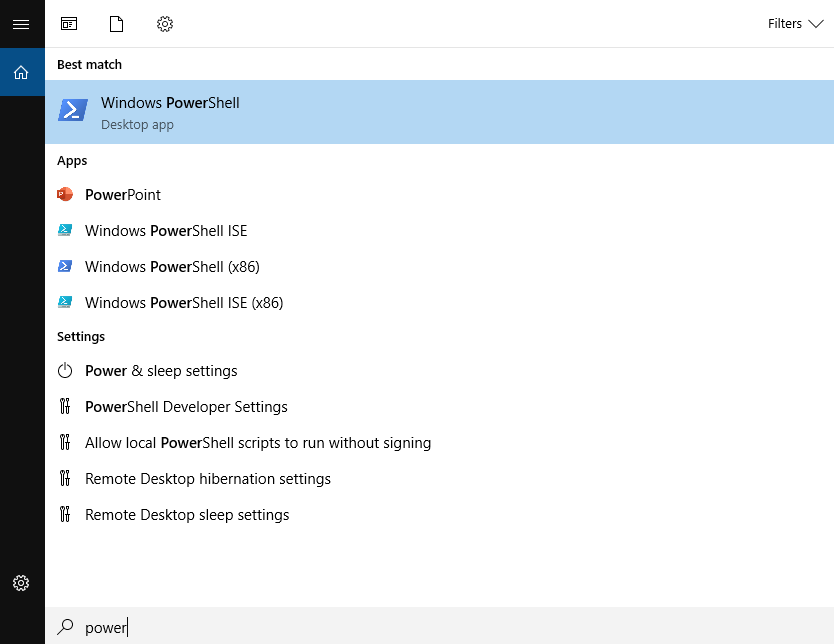


Try something!

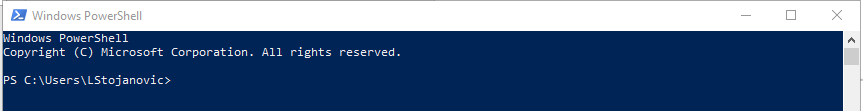


It works! How else can you do the same thing?

Search for “powershell”



Open it… you get an intimidating looking blue screen…



You can navigate to your project directory like this. Try running this line of code:

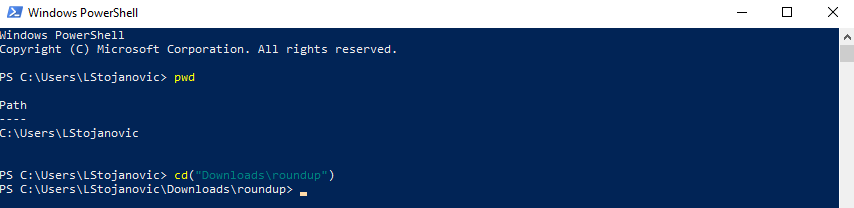
pwd

This stands for “print working directory.” The working directory is just the file path to the current folder the computer is looking at. Anything you tell it to do will be done from the working directory.

Now try:

cd(“your/path/here”)

This will change your working directory. I have set up a folder in the Downloads part of my computer called roundup. That is where I intend to do the coding for the Python webscrape project. To set up a folder for your project, you can just use the File Explorer, like you would for any other folder you’ve created on your computer.

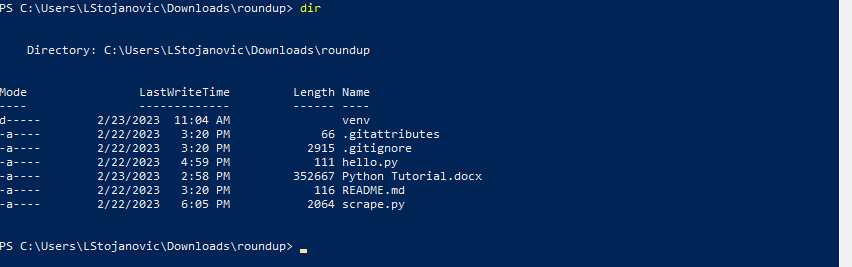


You can use “dir” to see if you already have a venv set up. The “dir” function shows everything that is in your current directory. Technically, a “venv” (virtual environment, more on that below) is just a special type of folder.

dir

Chances are, if you’re running dir on a newly minted folder you’ve just created, it will show nothing there! (That’s okay – you wouldn’t expect anything to show up in an empty folder!). If you want to try an experiment, copy a random file into your working directory. If you run “dir” again, it should show up!

Here’s the dir result from my roundup folder:



Look – I do have a venv!

What if you don’t have a venv and want to set one up? Let me show you with a new directory. It’s empty. It’s called “test\_directory”. As you can see, when I run “dir”, nothing shows up.



Here’s [more info on venv](https://docs.python.org/3/library/venv.html). As far as I can tell, it comes with Python. No need to install any packages.

To make a new venv type, run the following code:

python -m venv /path/to/new/virtual/environment

so in this case, since I am already in the desired venv directory, I’ll make the /path/to/new/venv/ just the following: venv

(This is standard practice.)

Don’t worry if it’s a bit slow to run, that is expected.

The directions from above apply. To create a venv, type:

python -m venv name\_of\_venv

Standard practice says that people name their virtual environments “venv”. So try running the following line of code:

python -m venv venv



Now that we’ve created the venv, let’s activate it! Type the following:

name\_of\_venv\Scripts\activate

if you named your venv “venv”, then the code should be:

venv\Scripts\activate



You’ll know if you properly opened the venv because it will be in green. Perfect!



To leave the venv, you can either close PowerShell, or you can run the following line of code:

deactivate



Nice job! 😊

What if you want to open IDLE from the venv? Here’s a [helpful stackoverflow](https://stackoverflow.com/questions/4924068/how-to-launch-python-idle-from-a-virtual-environment-virtualenv#:~:text=After%20running%20the%20IDLE%2C%20you,using%20ctrl%2Bo%20keyboard%20shortcut.&text=Save%20this%20answer.,-Show%20activity%20on) where I got the answer from:

Step 1. Open the venv.

Step 2: run the following: python -m idlelib.idle

e.g.



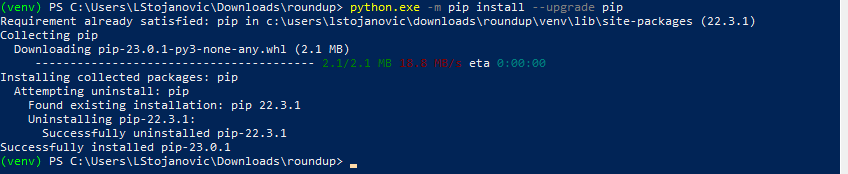
Wow!

How do we know that worked? Well, one way is we can try to install packages into the venv.

Follow these directions to install packages in a virtual environment. (Based in part upon [this](https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/) link.)

Open your directory in question and your venv.

Then run the following: python.exe -m pip install --upgrade pip (you only need to do this once)



You can check what version you have:

python -m pip --version

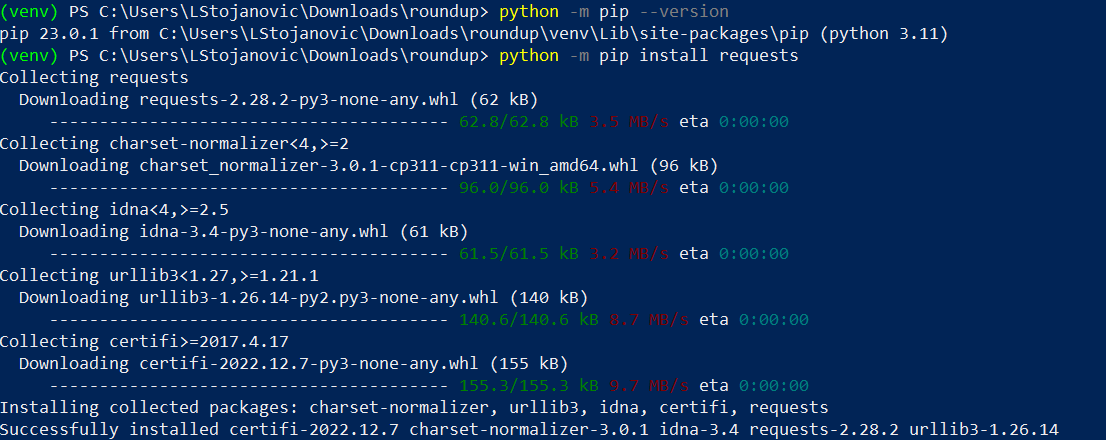


Cool!

Pip is a special type of package in R. I don’t understand exactly what it does, but most Python users use it as a package manager. So, in order to install packages, you want to install pip. There’s definitely some more advanced stuff and nuance having to do with pip, and some people don’t use it, but that is beyond the scope of this tutorial document (and my own knowledge).

Now that you have pip installed, we can try to install a brand-new package. Try running the following code. In this case, the package we are installing is called “requests”:

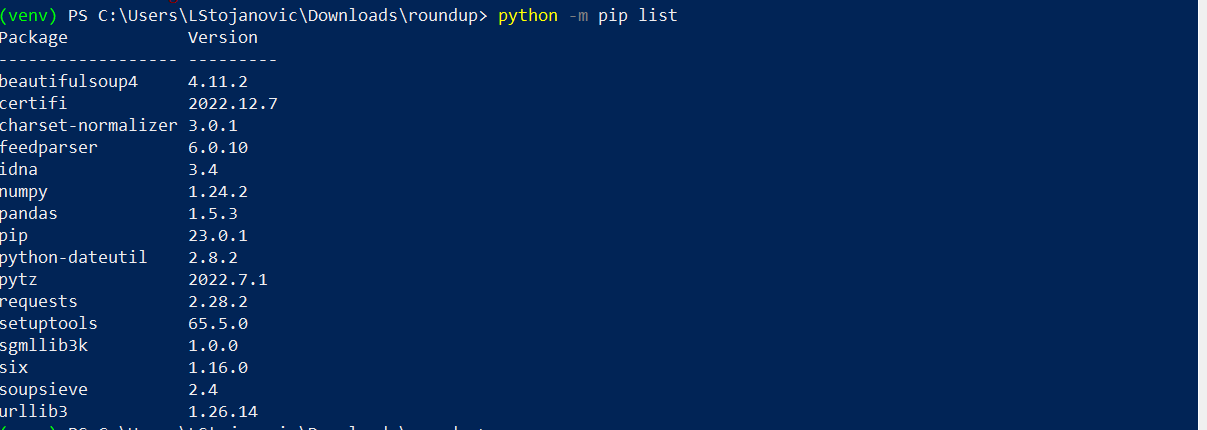
py -m pip install requests



You can do this to successfully install packages! Try installing a few that you think you’ll need. Then come back to this guide. I will show you how to check that they have been properly loaded into your computer.

Check this out! What packages have I installed? To see, type in the following code:

python -m pip list



Wow! Look at all those packages! Note that you can use instructions linked [here](https://docs.python.org/3/tutorial/venv.html#managing-packages-with-pip) to download specific versions of packages that aren’t the most recent ones, if you want.

Let’s now try creating our first ever requirement.txt file. This is essential for sharing our work with others. Packages are very big, and come with many versions. Sometimes, when packages are updated, old code that we wrote based on the packages can break (the ability for a script to continue to work over time despite these changes is called “backward compatibility” – or is it “forward compatibility”?). The issue of compatibility is one reason we use virtual environments. Virtual environments preserve the packages exactly as they were when we used them. Note that virtual environments do *not* preserve individual versions of Python. That is something that pyenv does. (Note that I think it would be difficult to use pyenv on Brookings computers, so I will not discuss it in this guide).

But before that, a note on uninstalling packages from https://pip.pypa.io/en/latest/user\_guide/#requirements-files :

**Uninstalling Packages**

pip is able to uninstall most packages like so:

py -m pip uninstall SomePackage

pip also performs an automatic uninstall of an old version of a package before upgrading to a newer version.

**What is a requirements.txt file?**

A requirements.txt file is an excellent resource for code replicability and sharing. It’s a simple text file listing of all the packages and versions of those packages that your project uses. Making one is very simple. And using one to load packages is also simple.

**Why use a requirements.txt file? Why not just share the whole virtual environment with someone else?**

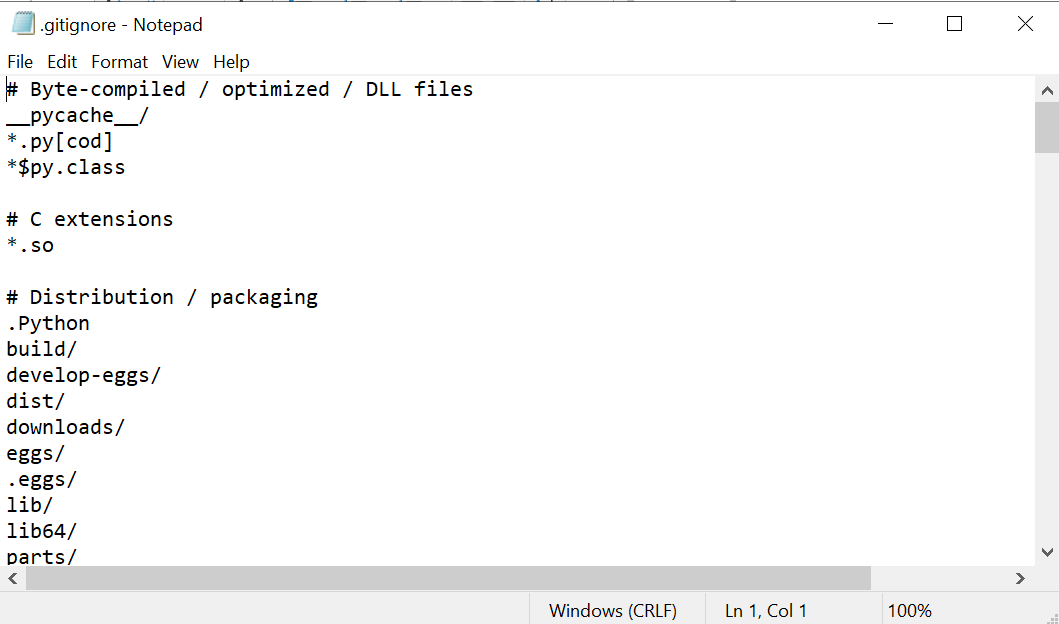
Good question. I don’t exactly know all the details, so if you want to know why, you might need to do more reading. If a simple answer will suffice, there are two main reasons that I understand:

1. Virtual environments have a lot of data and files. Uploading them and downloading them repeatedly is very impractical. When I’ve deleted sample virtual environments that I created for screenshots in this document, they took a decent amount of time to delete. Even with only 10 packages or so, the venv folders contained over 7,000 files! Hefty!
2. I don’t know the details of this point, but virtual environments are configured specifically for your computer. If you try to just send someone else your venv directory, it probably wouldn’t work properly on their computer, anyway!

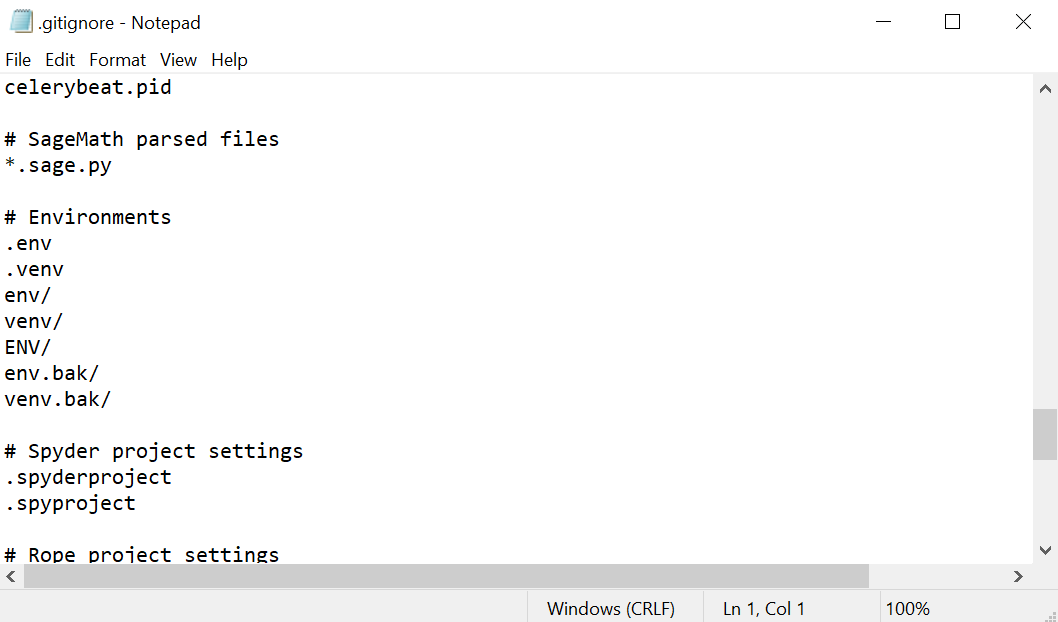
**Ok, I understand why I shouldn’t upload and download my venv constantly. But how do I stop Git from doing this for me?**

Good thing you asked. You can create a special type of file called .gitignore

I know that “.gitignore” seems like a scary file extension, but don’t worry! It’s also just a text file at heart. Here’s the current .gitignore file associated with the roundup project as it stands:



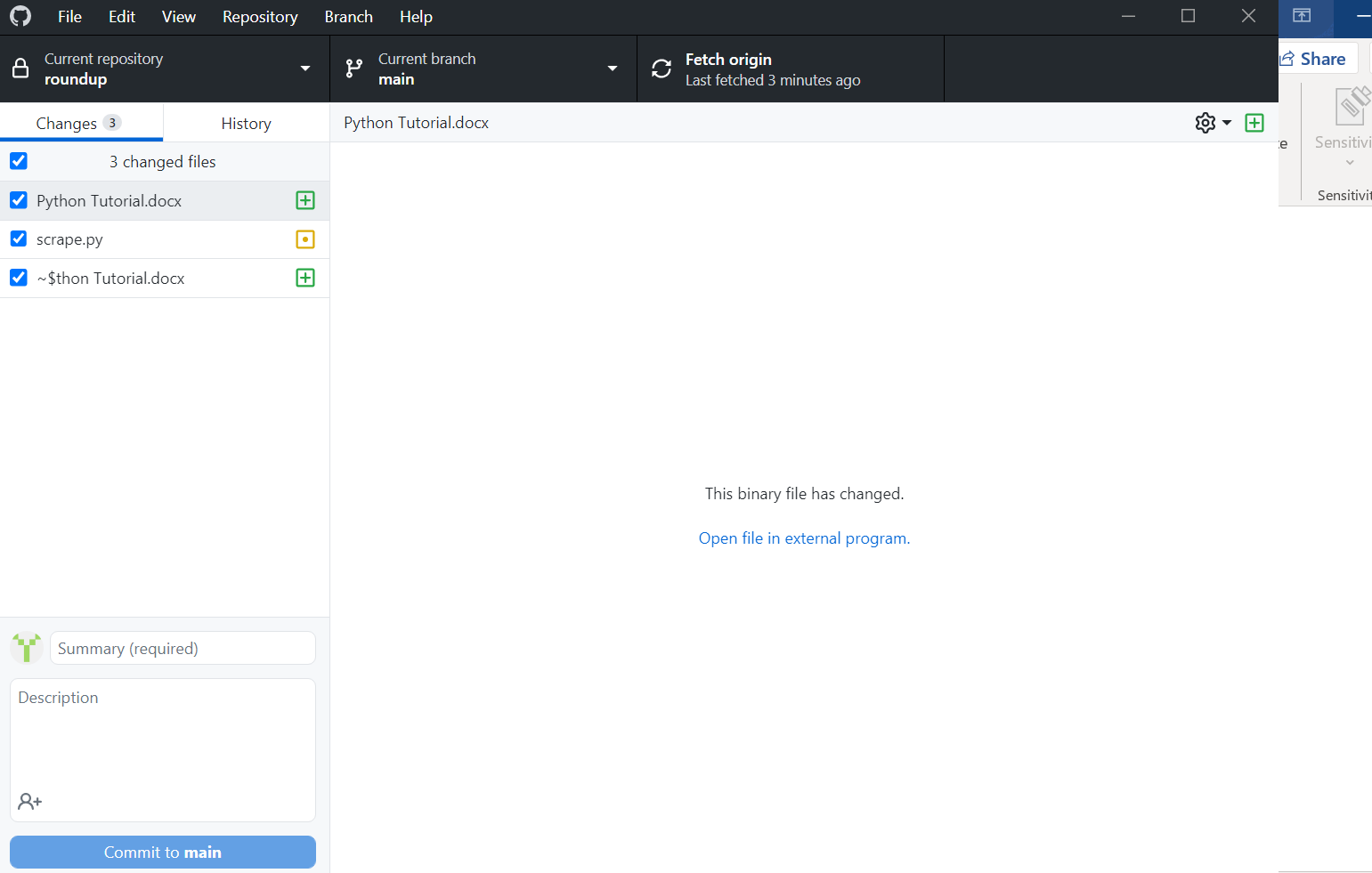
I haven’t scrolled all the way down. There’s more stuff at the bottom, that you might actually recognize:



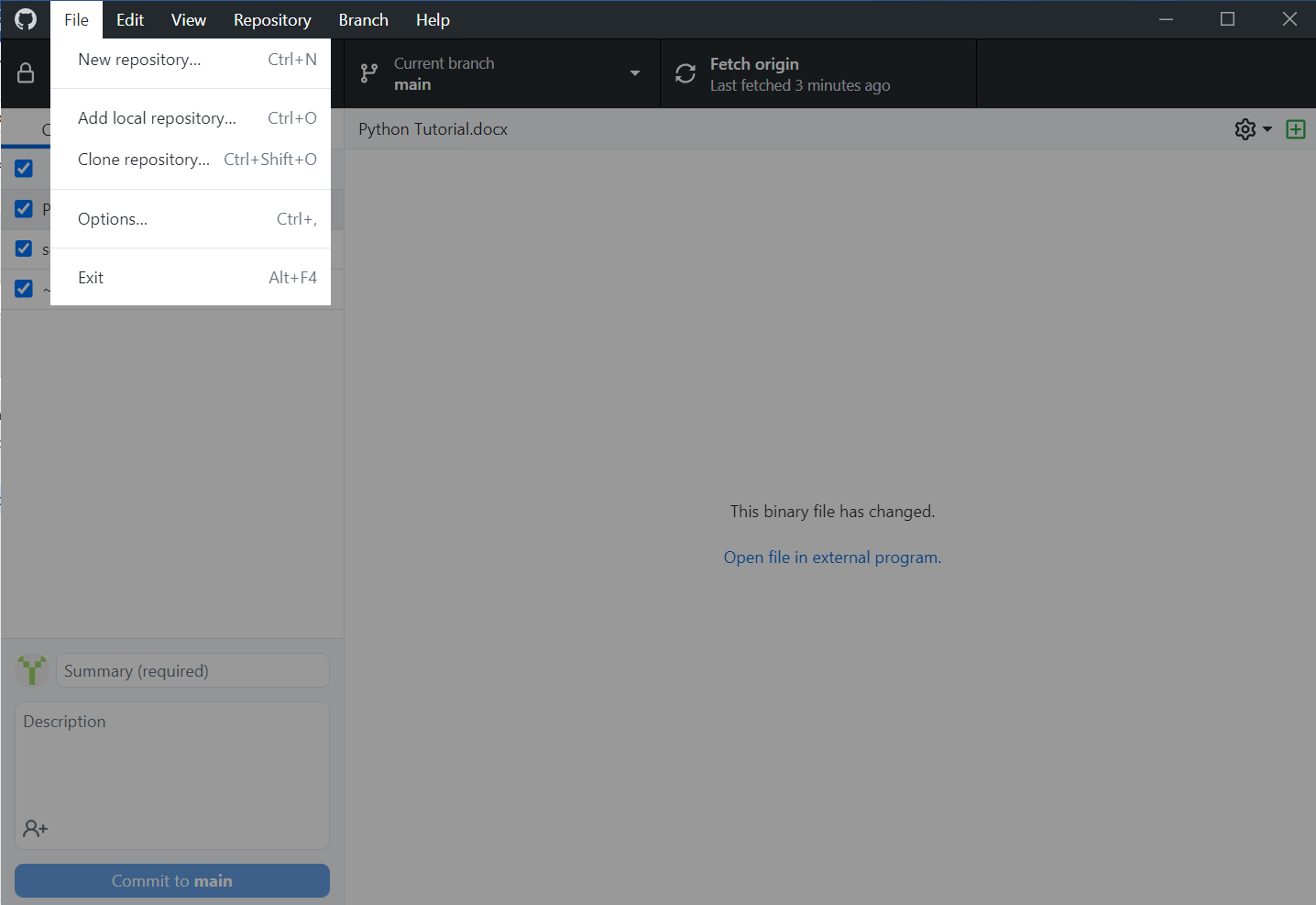
See the environments section? It’s telling GitHub that every time it uploads a file, it should ignore (i.e., not upload, download, or touch in any way) the files or directories starting with this extension. You can see “venv/” is on this list. Our virtual environment is safe from being uploaded and downloaded – a good sign.

How did I make this file? It was simple. Let me show you the step-by-step process, so that if you ever create your own project directory, you know what to do.

1. Open your GitHub for desktop app. Here’s what mine looks like:



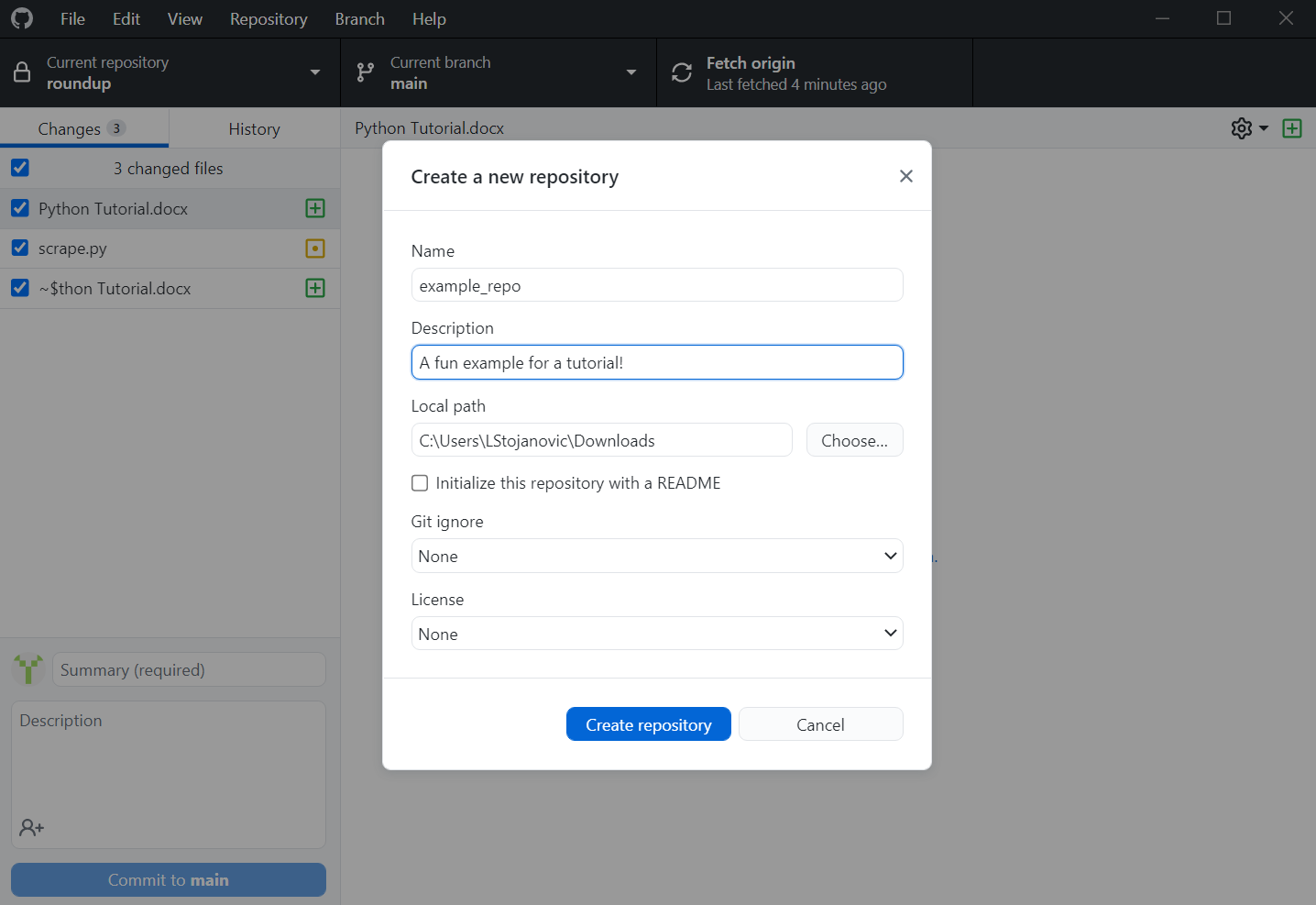
1. Click “File > New repository”



1. Give your repository a funky name and choose a file path. Add a fun description if you’d like!

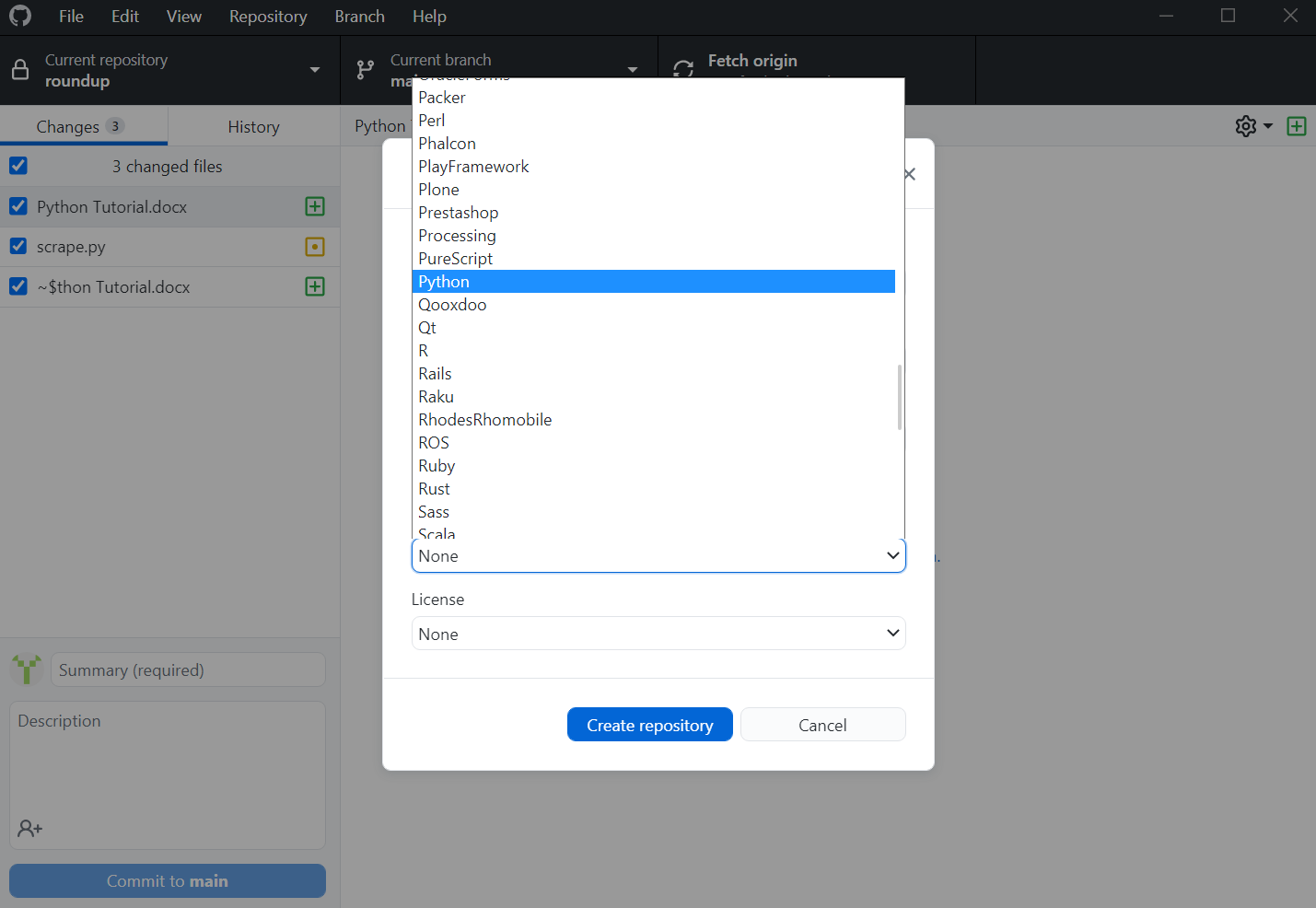
Note on the file path: You can save your directory anywhere on your computer that you like. However, if that location on your computer is already covered by a cloud service like OneDrive, SharePoint, or Google Drive, you might run into some issues. These services already do something similar to GitHub: They save directories to the cloud for easy storage and backup. Because of this, they may override the changes that GitHub makes. Say, for example, you save a file locally on your computer in the example\_repo. Then, you decide to push that change to the origin version of the file. Then it might be the case that SharePoint will locally override your changes by syncing. (I don’t know exactly how this works. It’s never happened to me, but I’ve been warned by Nasiha.) Anyway, just don’t do it.

My repo is saved in the “Downloads” folder in my computer. Cringey, I know! But that is one of the few areas of this computer that is not covered by a cloud syncing service.



1. The important step!!! See the setting called Git ignore? Select an item from the drop-down list. I don’t know exactly what the list means. I’ve tried Googling it, to no avail. But I figure it provides some examples of useful Git ignore files depending on the coding language you are using.

Don’t fret if you are confused! Just choose the “Python” version.



The .gitignore file is just a text file at heart. If you ever want to edit it, just use an application like Notepad to open it, like I did above. For our purposes, I think the only line of code you’ll need in it is one that says

venv/

But I do appreciate that they have fancy versions already set up. It looks like they even have versions that are pre-set for R.

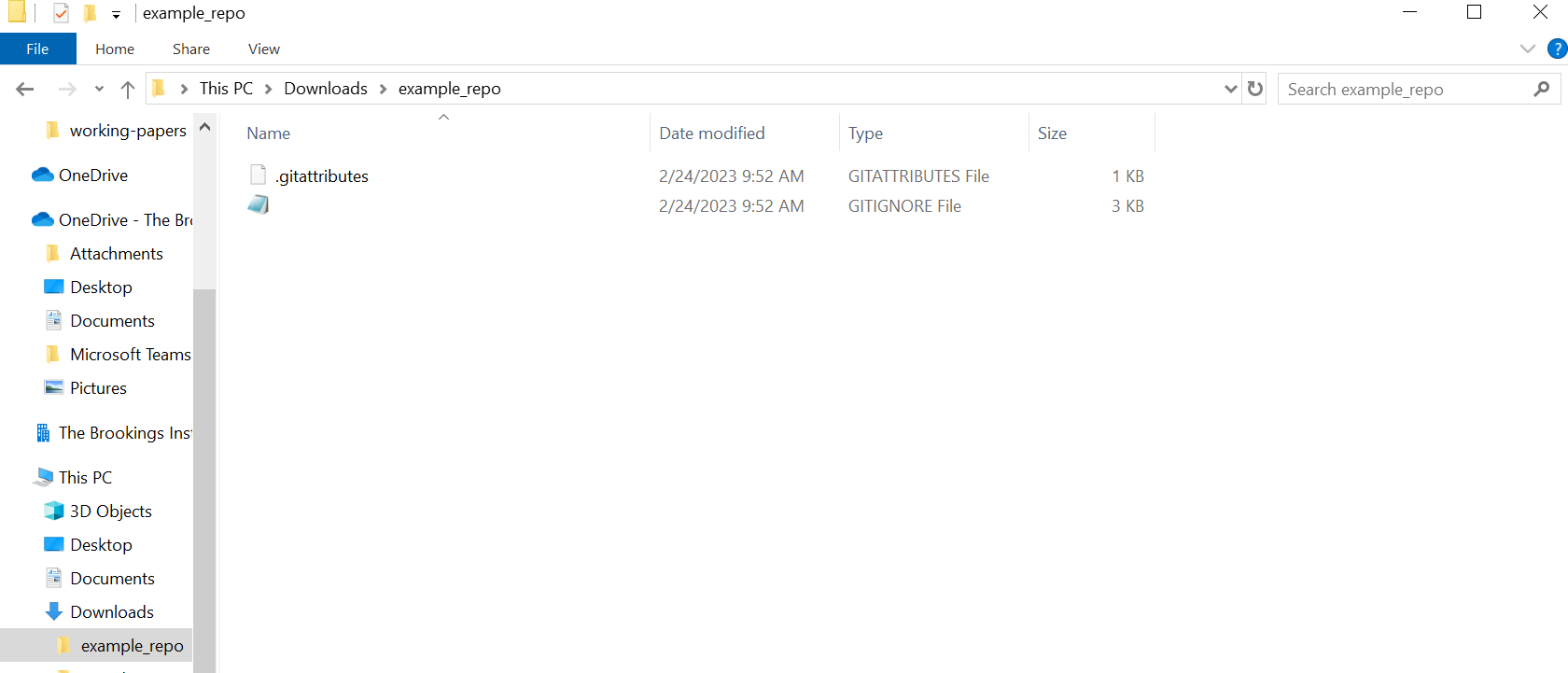
You can imagine how this might come in handy for other projects. I’ve tried to upload a large project on R to GitHub. My upload was cancelled because some of my raw data files exceeded the maximum file size. Instead of uploading the data files to GitHub, you can imagine putting them in your .gitignore file: e.g. by saving a directory in your project folder called “input”, and then adding a line to your .gitignore file that says:

Input/

Then, your readme file for the project could include a description of what the data sources are and where they can be found. This keeps your project light and portable.

See? The .gitignore file is a useful tool.

1. Here’s a screenshot of my File Explorer, just to show you the result. GitHub created the folder where I told it to, and also put a file in it called “.gitignore”, just like in the other example. Yay!



What is next? Before we invest a lot of time setting up our custom environment for this project, we want to make sure everything is neat and tidy. Are our packages saving to the right places? Could someone replicate our virtual environment? That is where the requirements.txt file comes into play.

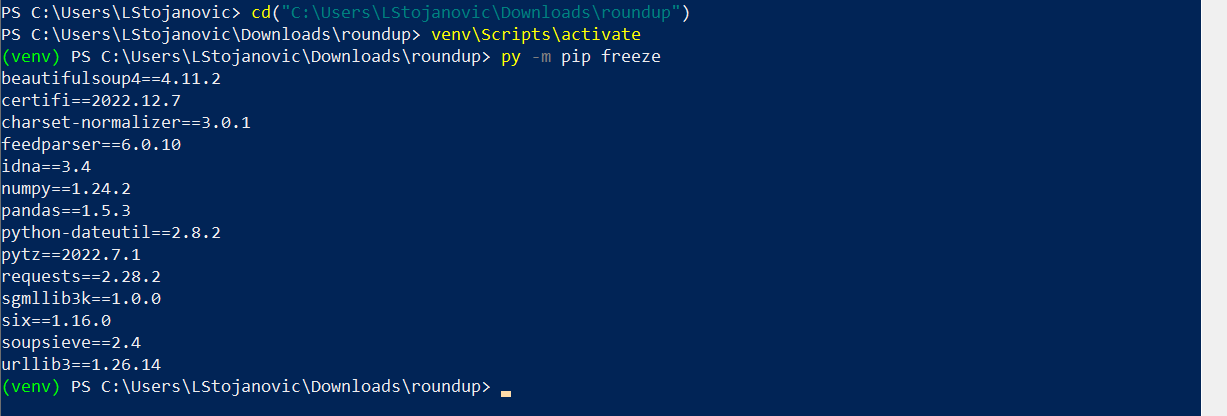
As I alluded to above, the requirements.txt file will tell a user what packages are needed, so that they can set up their own virtual environment for the project.

Let’s see if our virtual environment in our new project is replicable.

To create requirements.txt, do the following:

1. First, open your directory and activate your virtual environment (directions appear earlier in this document). Then, run the following line of code:

py -m pip freeze



This tells your computer to show whatever versions of the packages there are available at that moment in your current environment. Here, you can see the packages and versions associated with the roundup project’s virtual environment.

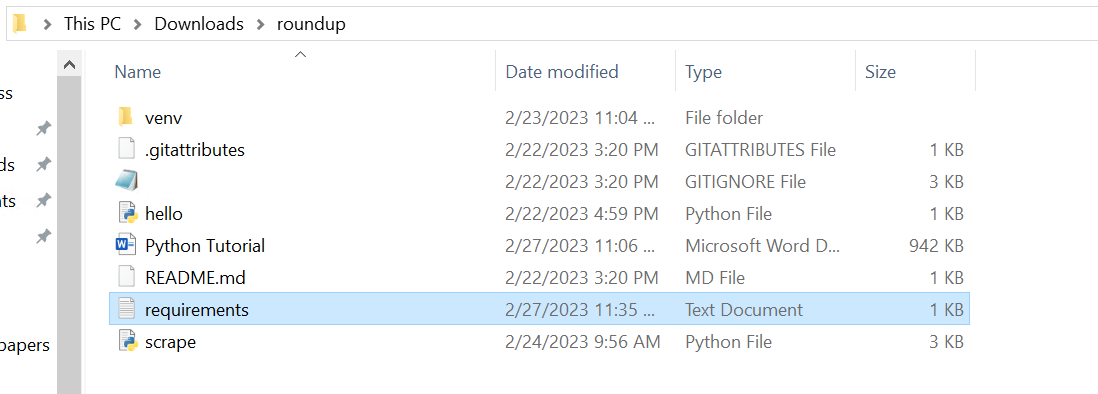
1. Now, let’s turn those into a usable requirements.txt file. Run this line of code:

py -m pip freeze > requirements.txt

Note: You didn’t really need to run the code in step 1; you could have just skipped to this step. But what I like about showing pip freeze first is that you see what packages there are before generating the requirements.txt file.

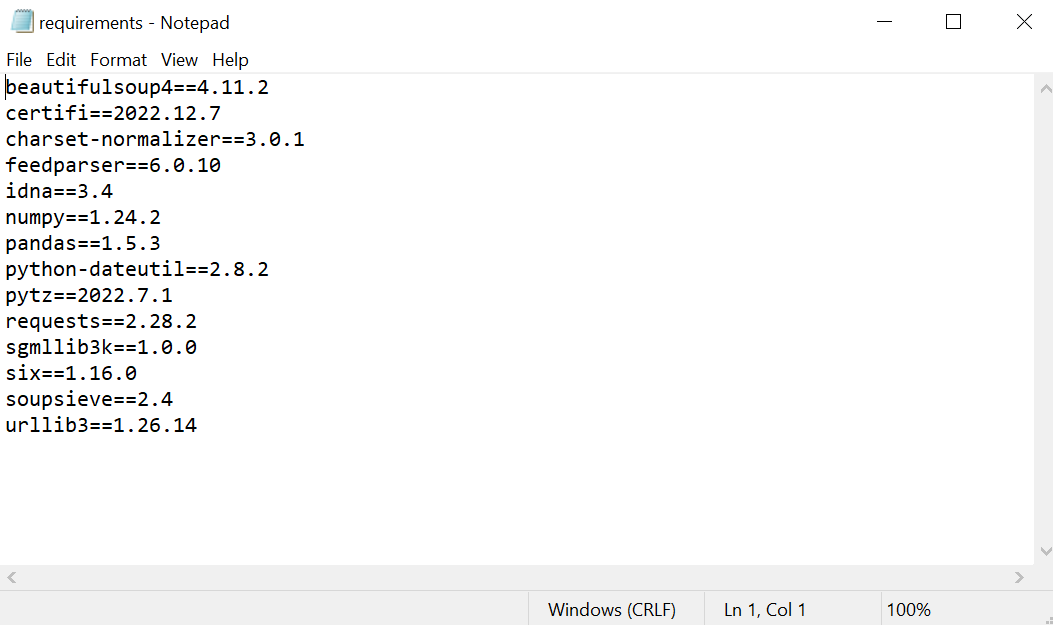


After running this line of code, you can see whether it worked. Navigate to your working directory and check if requirements.txt was created.



Success!

Let’s open it and check the contents.



Success!

Other resources:

<https://pip.pypa.iodef/en/latest/user_guide/#requirements-files>

[12. Virtual Environments and Packages — Python 3.11.0 documentation](https://docs.python.org/3/tutorial/venv.html#managing-packages-with-pip)

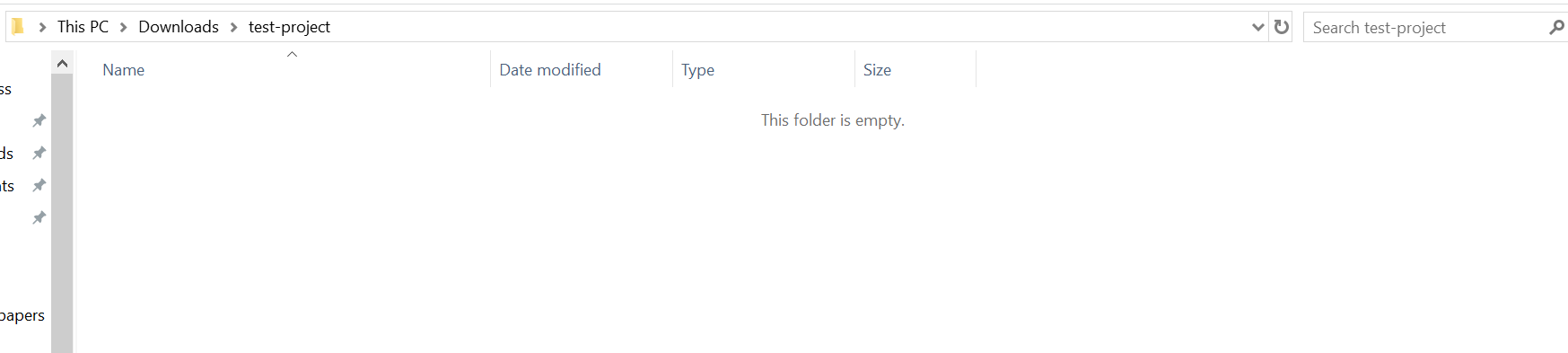
<https://stackoverflow.com/questions/31684375/automatically-create-requirements-txt>

Let’s check if our requirements.txt works. To do this step, I’m going to create a new project and an empty venv. This demonstration will explain how, upon setting up a new project, you can replicate the venv.

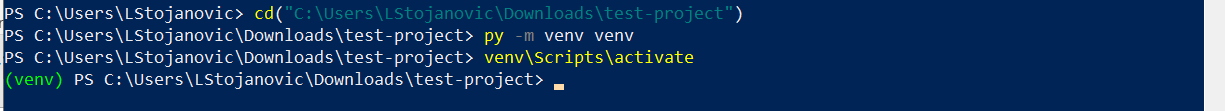
Note that for true replicability, you may want to also consider Python versioning, a capability that can be controlled using “pyenv”. But this is outside of the scope of the document.

To install packages in a requirements.txt file:

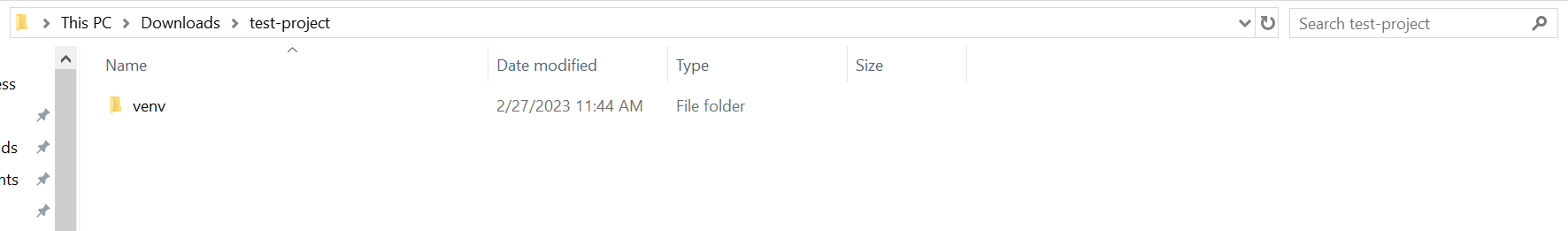
1. For this demonstration, I’ve made a new folder called test-project. Here, you can see that it is empty:



First, I will navigate to it in PowerShell and set up a venv in it, as in my prior demonstrations. Then I activate the venv.

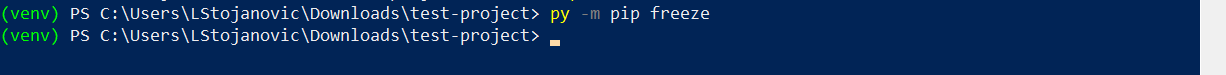


Navigating back to the folder in File Explorer, you can see that the venv now exists. Perfect.



1. Let’s first demonstrate that we don’t have any packages yet. Let me run

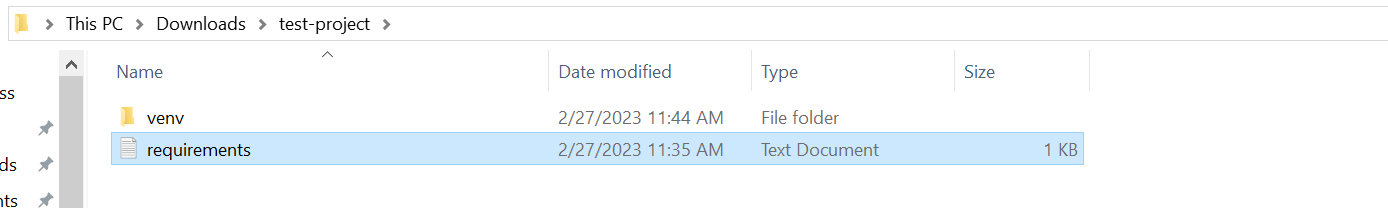
py -m pip freeze



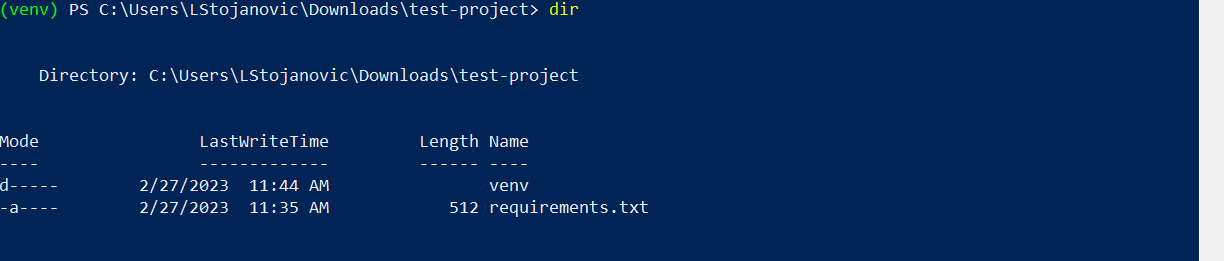
As you can see, nothing happened! That is because there are no packages yet in this venv.

1. Now, let’s get the requirements.txt file. For the purposes of this demonstration, I am doing nothing fancy. All I’m going to do is copy and paste the requirements.txt file I just made in the previous demonstration and put it in this folder.

What is more likely to happen in the future is that you will fork (or fetch? I’m not sure… I’m not super familiar with the lingo) a GitHub repo on to your hard drive. Then, you’ll want to set up the venv. The requirements.txt will probably already be there in the working directory (if you’re working with any respectable code…)



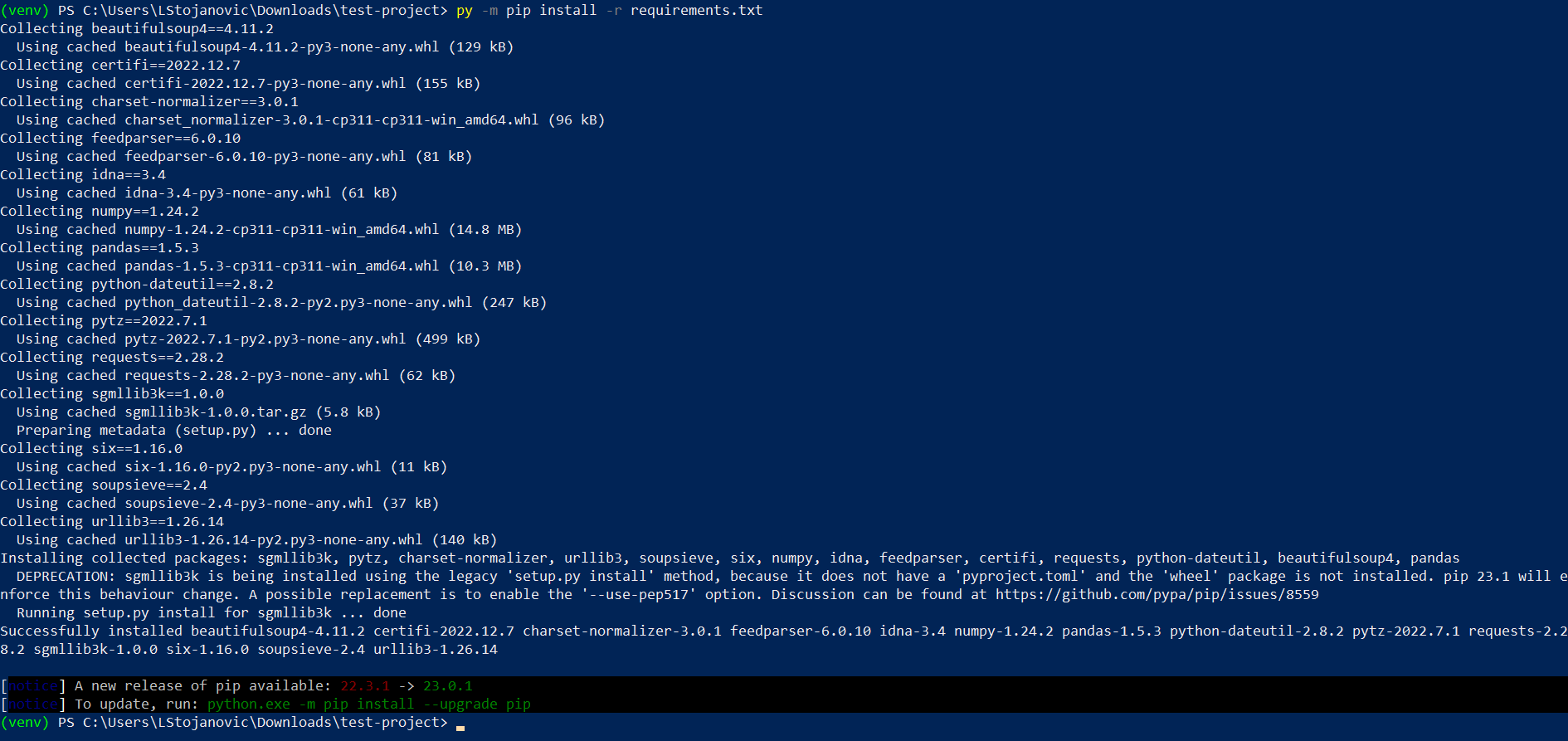
There! I copied and pasted it in. See?



1. Now, let’s unpack the files in requirements.txt! Run the following code:

py -m pip install -r requirements.txt

It might take a while if there are a lot of packages. Maybe grab yourself a cup of tea while you wait.



Wow! Look at how much it installed. It’s even telling me there’s a new version of pip. (Since this is just a demo, I won’t install it. But you bet that in the main roundup folder, I will.)

1. Let’s just confirm that it did what it said it did. Run:

py -m pip freeze



Perfect! As you can see, it installed everything from requirements.txt seamlessly.

Keep this workflow in mind as you work on your next project. It may be a good idea to update requirements.txt every now and then in order to ensure that your work will be replicable on a different computer.

Anyway, you now have the basics to ensure that your new Python adventure is tidy and replicable. Good luck, and enjoy!