Learn to use Click (Why? So you can use decorators to mark your functions as commands SO YOUR FUNCTIONS BECOME COMMANDS – ***COMMANDS THAT ARE CALLED AT THE COMMAND LINE! COOL!)

Make a directory
Create a virtual environment in that directory
Virtualenv creates isolated python environments for python libraries
Virtualenv makes an env directory & puts a special copy of the python exectuable in env/bin
python3-venv is the ubuntu equivalent of Python3 venv which is similar to virtualenv
Install setuptools which is a tool that not only builds application packages but also uploads & installs them nb works v well with virtualenv & pip
Activate the virtual environment
Now let's create a proper python package
So, make a setup.py
Specify package name, version, module to be installed, dependencies like click itself, entry points.

What are entry points? Code objects derived from setuptools that register something with a specific key in one package that another package can query for. Entry points are registered under console-scripts in setup.py as a list that setuptools can use. These console scripts are instructions to setup tools to associate some other metadata with our python package, used internally to create cli's,*** or to use other libraries

Within console scripts create a command line executable that points to our module and the cli function

Now make the cli module a function***that starts as

```
def cli():
  do some other stuff e.g., print("Hello World")
```

& this together with setup.py is sufficient to install the python package

Python loads source code file (module)
Python defines special variables
Python defines _name_

In the case where this is the main program, i.e., the file run by a user at a prompt, then _name_ is set to '_main_' and blocks A & B will both be run when running my_code.py from the command line. However, only block A runs if I have imported a module, e.g., that_guys_code.py with an import statement because _name_ is set to '_that_guys_code_' rather than '_main_' & what gets called is code contained within that other module

```
<Block A>
if __name__ == '__main__':
    main()
      <Block B>
```

The top line e.g., if_name == '_main_': is the interpreters '*where am I?*' test to determine if it is running on the code it is looking at (parsing), or if it is actually 'peeking' into another file. It only allows the main() to run if main() is the primary entry point but if a module has been imported then it doesn't because the entry point is within the module import and calls on functions & classes, will be made on the other module. This gives the programmer flexibility to make code behave differently if called externally