Creating a Python Makefile

10 minute read Updated: July 8, 2021

Aniket Bhattacharyea

Even though Python is regarded as an interpreted language and the files need not be compiled separately, many developers are unaware that you can still use `make` to automate different parts of developing a Python project, like running tests, cleaning builds, and installing dependencies. It's honestly an underutilized function, and by integrating it into your routine, you can save time and avoid errors.

`make` is a commonplace tool in the world of software development, especially compiled languages like C or C++. It is a tool which controls the generation of executable and other non-source files from a program's source file. It can automate the process of building software by tracking its dependencies and compiling the program only when the dependencies change.

The reason `make` is very common with compiled languages is because the compilation commands for those languages can be long and complicated and difficult to remember. Also, you need to compile each file and link the resulting object files together. So, whenever one of the files changes, it becomes necessary to recompile it.

In this tutorial, you will learn the basics of `make` and how it can be used in a Python project.

Is Python Compiled or Interpreted?

Usually, all the programming languages can be classified into compiled or interpreted languages. In simple words, in a compiled language, the program is converted from a high level language to machine language. Whereas, in case of interpreted language, the source code is read and executed one line at a time.

Since in an interpreted language you cannot compile the source files independently beforehand, you cannot utilize `make` like you would do for a compiled language. This makes `make` relatively underutilized for an interpreted language.

Now, Python is usually considered to be an interpreted language. When you run a Python code, the Python interpreter reads the file line-by-line and runs it.

But behind-the-scenes, the source code is compiled into bytecode. These are similar to CPU instructions, but instead of being run by the actual CPU, these are executed by a software called a Virtual Machine (VM), which acts as a pseudo-microprocessor that runs the bytecodes. The advantage is that you can run Python on any platform as long as the VM is installed.

When you run a Python code, the interpreter implicitly compiles the code into bytecode and interprets it with the VM. The reason Python is regarded to be an interpreted language is because the **compilation** step is implicit. You don't have to invoke a compiler manually.

When you import a module into your code, Python compiles those modules into bytecode for caching purposes. These are stored in a directory named `__pycache__` in the current directory, which contains compiled `.pyc` files.

Although you cannot compile these modules using `make`, you can still use `make` for automation tasks like running tests, installing dependencies, cleaning the `.pyc` files etc.

Using Make With Python

In this tutorial, you'll create a simple app that makes requests to `http://numbersapi.com` and fetches random trivia about a user given number.

The Sample Source Code

First, you'll start by creating a file **`api.py`** with the following code:

```
import requests

def get_fact(number):
    url = "http://numbersapi.com/{}".format(number)

    r = requests.get(url)
    if r.status_code == 200:
        print(r.text)
    else:
        print("An error occurred, code={}".format(r.status_code))
```

Create another file **`app.py`** with the following code:

```
import api

api.get_fact(input("Enter a number: "))
```

This file simply imports the **`api`** module and calls the **`get_fact()`** function with a user provided number.

Finally, create a file **`requirements.txt`** with the dependencies of the app:

```
requests
```

This app only depends on the **`requests`** module, but a real life project possibly depends on a large number of modules. Let's run the app and see how it works.

First, install the dependencies:

```
pip install -r requirements.txt
```

Then, run the actual app:

```
python app.py
```

This will prompt you for a number and show you a trivia about that number:

```
20 is the number of questions in the popular party game Twenty Question
```

Now let's integrate `make` with our project to automate the installation of dependencies and running the app.

The Makefile

To use `make` in your project, you need to have a file named `Makefile` at the root of your project. This file instructs `make` on what to do. The Makefile consists of a set of rules. Each rule has 3 parts: a target, a list of prerequisites, and a recipe. They follow this format:

```
target: pre-req1 pre-req2 pre-req3 ...
    recipes
    ...
```

*Note that there are **tabs** before the recipe lists. Anything other than tabs will result in an error.*

The `target` represents a goal that you want to achieve, usually this is a file that needs to be created in your build. The prerequisites list tells `make` which files are this target dependent on. The prerequisites can be a file or another target. Finally the recipes are a list of shell commands that will be executed by make as part of building the target.

When `make` executes a target, it looks at its prerequisites. If those prerequisites have their own recipes, `make` executes them and when all the prerequisites are ready for a target, it executes the corresponding recipe for the current target. For each target, the recipes are executed only if the target doesn't exist or the pre-requisites are newer than the target.

Our app has two targets. First, the dependencies must be installed and then the app can be run.

Let's create the rule for running the app first:

```
run:
    python app.py
```

The target is named `run` and it has no prerequisites, which it will be run every time you run `make run`. You can test it by running `make run` in a terminal.

Create another target for the setup stage:

```
setup: requirements.txt
    pip install -r requirements.txt
```

The `setup` target depends on the `requirements.txt` file. Whenever the `requirements.txt` file changes, the dependencies will be refreshed by running `pip install -r`.

Finally, let's have a `clean` rule to clean up the `__pycache__` folder:

```
clean:
    rm -rf __pycache__
```

Creating a Virtual Environment

The sample app depends on the `requests` library only. However, in a large project, there might be numerous dependencies. And if you are running multiple apps, it's possible that some apps require the same dependencies, but a different version. This means that one Python installation may not be capable of satisfying the requirements of all applications. The solution for this is to use a **virtual environment**. This is a self-contained directory tree that contains a Python installation of a specific version.

Different apps can use their own virtual environment where they can install their requirements. The virtual environments are isolated from each other,

which means there will be no dependency conflicts.

Python provides a module called **venv** which is used to create and manage virtual environments. Let's see how a virtual environment can be used in the sample app.

First, you need to create a virtual environment in the project root:

```
python3 -m venv venv
```

This creates a **venv** folder in your current directory, which contains the necessary files to make the virtual environment.

There are two ways to use this virtual environment. Instead of using **python3** or **pip**, you have to use **./venv/bin/python3** or **./venv/bin/pip** to run the app or install dependencies:

```
python3 app.py # Uses the system Python
./venv/bin/python3 # USes the virtualenv Python
```

But writing **./venv.bin/** every time can be time consuming, especially if you have to run a lot of commands that use Python or pip. To overcome this, you can "activate" the virtual environment by running:

```
./venv/bin/activate
```

This will load the virtual environment in the current shell. This environment will stay active as long as you don't close the shell, or deactivate manually.

Once activated, running **python3** or **pip** will use the executables from the virtual environment. If you now run **pip install -r requirements.txt**, the modules will be installed in the **venv** directory.

Once you are done with the virtual environment, you can deactivate the environment by running the **deactivate** command.

venv in Make

You can utilize **make** to automatically refresh your virtual environment and run your app with this virtual environment. To automatically reinstall the

dependencies whenever the `requirements.txt` file changes, write the following in your Makefile:

```
venv/bin/activate: requirements.txt
 python3 -m venv venv
 ./venv/bin/pip install -r requirements.txt
```

Here the target is `venv/bin/activate` which depends on `requirements.txt`. Whenever `requirements.txt` changes, it rebuilds the environment and installs the dependencies with pip, which re-creates the `activate`. The actual goal here is the existence of the `venv` directory, but since `make` can only work with files, `venv/bin/activate` is used instead.

To run the app with this environment, create the following rule:

```
run: venv/bin/activate
 ./venv/bin/python3 app.py
```

The `run` target depends on the `venv/bin/activate` target. Once that target is satisfied, it runs the app using the virtual environment.

Finally, update the `clean` target to also delete the `venv` directory:

```
clean:
 rm -rf __pycache__
 rm -rf venv
```

Let's test this all out. First, delete the `venv` directory if you have one. Now run `make run`. Since the `venv/bin/activate` file does not exist, `make` will run the `venv/bin/activate` target, which will install the dependencies and finally run the app using the virtual environment.

```
→  make run
python3 -m venv venv
./venv/bin/pip install -r requirements.txt
Collecting requests
  Using cached requests-2.25.1-py2.py3-none-any.whl (61 kB)
Collecting idna<3,>=2.5
  Using cached idna-2.10-py2.py3-none-any.whl (58 kB)
Collecting certifi>=2017.4.17
  Using cached certifi-2021.5.30-py2.py3-none-any.whl (145 kB)
Collecting urllib3<1.27,>=1.21.1
  Using cached urllib3-1.26.5-py2.py3-none-any.whl (138 kB)
Collecting chardet<5,>=3.0.2
  Using cached chardet-4.0.0-py2.py3-none-any.whl (178 kB)
Installing collected packages: idna, certifi, urllib3, chardet, requests
Successfully installed certifi-2021.5.30 chardet-4.0.0 idna-2.10 requests-2.25.1 urllib3-1.26.5
WARNING: You are using pip version 20.2.3; however, version 21.1.2 is available.
You should consider upgrading via the '/home/aniket/make-python/venv/bin/python3 -m pip install --upgrade pip' command.
./venv/bin/python3 app.py
Enter a number: 10
10 is the number of Provinces in Canada.
```

`make run` for Python `venv`

If you run `make run` once again, only the app will be run and the virtual environment will not be refreshed. You can use the `touch` command to stimulate a change in the `requiremenst.txt` file which will cause `make` to run the setup step again -

```
touch requirements.txt
```

Now if you run `make run`, the virtual environment will be recreated and then the app will be run.

Using Variables

Observe that in our Makefile, we have references to the `venv` directory in multiple places. In future, if we want to change the directory name to something else, we have to remember to perform the change in all the places. Also there isn't any way for the user to customize the directory name without editing the Makefile. To overcome this, we can use variables. The variables not only make the Makefile cleaner, they can be overridden by the user without editing Makefile.

A variable in Makefile starts with a $ and is enclosed in parentheses () or braces {}, unless its a single character variable.

To set a variable, write a line starting with a variable name followed by `=`, `:=` or `::=`, followed by the value of the variable:

```
VENV = venv
```

Here the variable `VENV` is set to `venv`. Now whenever you use this variable in a rule, it will be replaced by its value.

The rules now can be rewritten using the `VENV` variable:

```
VENV = venv
PYTHON = $(VENV)/bin/python3
PIP = $(VENV)/bin/pip

run: $(VENV)/bin/activate
 $(PYTHON) app.py


$(VENV)/bin/activate: requirements.txt
 python3 -m venv $(VENV)
```

```
	$(PIP) install -r requirements.txt

clean:
	rm -rf __pycache__
	rm -rf $(VENV)
```

I have also replaced references to `python3` and `pip` with a variable. By default, they will use the binaries from the virtual environment.

The variables can be overridden by providing the values when running the make command:

```
make VENV=my_venv run
```

Using `VENV=my_venv` overrides the default value of `VENV` and now the virtual environment will be created in `my_venv` directory. It is a good practice to use variables for all commands used in the Makefile, as well as their options and directories. This provides an easy way for the user to substitute alternatives.

Phony Targets

Your Makefile contains two special targets - run and clean. Special in the sense they don't represent an actual file that exists. And since `make` executes the recipes of a target if that target does not exist, these two targets will always be executed. However, if later if you have a file called `run` or `clean`, then since these targets have no prerequisites, `make` will consider these to always be newer and so, will not execute the recipes.

To overcome this, `make` has something called a Phony target. By declaring a target to be Phony, you tell `make` not to consider an existing file with the same name. To make the `run` and `clean` targets phony, add this to the top:

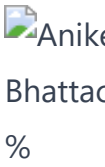
```
.PHONY: run clean
```

Conclusion

Using `make` in your Python projects opens the door to lots of possibilities in terms of automation. You can use `make` to run linters like `flake8`, run tests

using `pytest`, or run code coverage using `coverage`. If you wish to learn all the features of `make`, be sure to check out the **manual** by GNU.

Being full of features, and primarily targeted towards compiled languages, `make` can be difficult to use in a Python project, especially in large ones. If you are looking for a solution to avoid the complexities of Makefile, while still retaining the powerful automation capabilities, check out **Earthly**. Earthly takes a modern approach towards project builds and combines the best of both Makefile and Dockerfile. Earthly provides understandable and repeatable build scripts without any complexity.



Aniket Bhattacharyea
Aniket is a student doing a Master's in Mathematics and has a passion for computers and software.

Published: July 8, 2021

Get notified about new articles!

Email Address

Subscribe

We won't send you spam. Unsubscribe at any time.

You may also enjoy

Stop saying 10x developer
6 minute read

Here is part of a rather infamous Twitter thread: 10x engineers rarely look at help documentation of classes or methods. Given a product feature, they ca...

Bash String Manipulation

6 minute read

One thing that bash is excellent at is manipulating strings of text. If you're at the command line or writing a small script, then knowing some bash string...
