A Beginner-friendly Flake Tutorial

April 25, 2023

Amy de Buitléir

Contents

Hello, flake!

Before learning to write Nix flakes, let's learn how to use them. I've created a simple example of a flake in this git repository. To run this flake, you don't need to install anything; simply run the command below. The first time you use a package, Nix has to fetch and build it, which may take a few minutes. Subsequent invocations should be instantaneous.

```
$ nix run "git+https://codeberg.org/mhwombat/hello-flake"
Hello from your flake!
```

That's a lot to type every time we want to use this package. Instead, we can enter a shell with the package available to us, using the `nix shell` command.

```
$ nix shell "git+https://codeberg.org/mhwombat/hello-flake"
```

In this shell, the command is our `$PATH`, so we can execute the command by name.

```
$ hello-flake
Hello from your flake!
```

Nix didn't *install* the package; it merely built and placed it in a directory called the "Nix store". Thus we can have multiple versions of a package without worrying about conflicts. We can find out the location of the executable, if we're curious.

```
$ which hello-flake
/nix/store/0xbn2hi6h1m5h4kc02vwffs2cydrbc0r-hello-flake/bin/hello-flake
```

Once we exit that shell, the `hello-flake` command is no longer available.

```
$ exit
$ hello-flake
sh: line 3: hello-flake: command not found
```

Actually, we can still access the command using the store path we found earlier. That's not particularly convenient, but it does demonstrate that the package remains in the store for future use.

```
/nix/store/0xbn2hi6h1m5h4kc02vwffs2cydrbc0r-hello-flake/bin/hello-flake
```

The hello-flake repo

Let's clone the repository and see how the flake is defined.

```
$ cd ~/tutorial-practice
$ git clone https://codeberg.org/mhwombat/hello-flake
Cloning into 'hello-flake'...
$ cd hello-flake
$ ls
flake.lock
flake.nix
hello-flake
LICENSE
README.md
```

This is a simple repo with just a few files. Like most git repos, it includes `LICENSE`, which contains the software license, and `README.md` which provides information about the repo.

The `hello-flake` file is the command we were executing earlier. This particular executable is just a shell script, so we can view it. It's an extremly simple script with just two lines.

```
$ cat hello-flake
#!/usr/bin/env sh

echo "Hello from your flake!"
```

Now that we have a copy of the repo, we can execute this script directly.

```
$ ./hello-flake
Hello from your flake!
```

Not terribly exciting, I know. But starting with such a simple package makes it easier to focus on the flake system without getting bogged down in the details. We'll make this script a little more interesting later.

Let's look at another file. The file that defines how to package a flake is always called `flake.nix`.

```
$ cat flake.nix
{
  description = "a very simple and friendly flake";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
  };

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = import nixpkgs { inherit system; };
      in
      {
        packages = rec {
          hello = pkgs.stdenv.mkDerivation rec {
            name = "hello-flake";

            src = ./.;

            unpackPhase = "true";

            buildPhase = ":";

            installPhase =
              ''
                mkdir -p $out/bin
                cp $src/hello-flake $out/bin/hello-flake
                chmod +x $out/bin/hello-flake
              '';
          };
          default = hello;
        };

        apps = rec {
       hello = flake-utils.lib.mkApp { drv = self.packages.${system}.hello; };
          default = hello;
        };
      }
    );
}
```

If this is your first time seeing a flake definition, it probably looks intimidating. Flakes are written in a functional language called Nix[1]. Yes, “Nix” is the name of both the package manager and the language it uses. We’ll look at this in more detail shortly. For now, I’d like to focus on the inputs section.

```
inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
  };
```

There are just two entries, one for `nixpkgs` and one for `flake-utils`. The first one, `nixpkgs` refers

[1] For an introduction to the Nix language, see Nix language basics.

to the collection of standard software packages that can be installed with the Nix package manager. The second, `flake-utils`, is a collection of utilities that simplify writing flakes. The important thing to note is that the `hello-flake` package *depends* on `nixpkgs` and `flake-utils`.

Finally, let's look at `flake.lock`, or rather, just part of it.

```
$ head -n 40 flake.lock
{
  "nodes": {
    "flake-utils": {
      "inputs": {
        "systems": "systems"
      },
      "locked": {
        "lastModified": 1681202837,
        "narHash": "sha256-H+Rh19JDwRtpVPAWp64F+rlEtxUWBAQW28eAi3SRSzg=",
        "owner": "numtide",
        "repo": "flake-utils",
        "rev": "cfacdce06f30d2b68473a46042957675eebb3401",
        "type": "github"
      },
      "original": {
        "owner": "numtide",
        "repo": "flake-utils",
        "type": "github"
      }
    },
    "nixpkgs": {
      "locked": {
        "lastModified": 1681665000,
        "narHash": "sha256-hDGTR59wC3qrQZFxVi2U3vTY+r02+Okbq080hO1C4Nk=",
        "owner": "NixOS",
        "repo": "nixpkgs",
        "rev": "3a6205d9f79fe526be03d8c465403b118ca4cf37",
        "type": "github"
      },
      "original": {
        "owner": "NixOS",
        "repo": "nixpkgs",
        "type": "github"
      }
    },
    "root": {
      "inputs": {
        "flake-utils": "flake-utils",
        "nixpkgs": "nixpkgs"
      }
```

If `flake.nix` seemed intimidating, then this file looks like an invocation for Cthulhu. The good news is that this file is automatically generated; you never need to write it. It contains information about all of the dependencies for the flake, including where they came from, the exact version/revision, and hash. This lockfile *uniquely* specifies all flake dependencies, (e.g., version number, branch, revision, hash), so that *anyone, anywhere, any time, can re-create the exact same environment that the original developer used.*

No more complaints of "but it works on my machine!". That is the benefit of using flakes.

Flake structure

The basic structure of a flake is shown below.

```
{
  description = ... # package description
  inputs = ... # dependencies
  outputs = ... # what the flake produces
  nixConfig = ... # advanced configuration options
}
```

The `description` part is self-explanatory; it's just a string. You probably won't need `nixConfig` unless you're doing something fancy. I'm going to focus on what goes into the `inputs` and `outputs` sections, and highlight some of the things I found confusing.

Inputs

This section specifies the dependencies of a flake. It's an *attribute set*; it maps keys to values.

To ensure that a build is reproducible, the build step runs in a *pure* environment with no network access. Therefore, any external dependencies must be specified in the "inputs" section so they can be fetched in advance (before we enter the pure environment).

Each entry in this section maps an input name to a *flake reference*. This commonly takes the following form.

```
NAME.url = URL-LIKE-EXPRESSION
```

As a first example, all (almost all?) flakes depend on "nixpkgs", which is a large Git repository of programs and libraries that are pre-packaged for Nix. We can write that as

```
nixpkgs.url = "github:NixOS/nixpkgs/nixos-VERSION";
```

where `NN.MM` is replaced with the version number that you used to build the package, e.g. `22.11`. Information about the latest nixpkgs releases is available at https://status.nixos.org/. You can also write the entry without the version number

```
nixpkgs.url = "github:NixOS/nixpkgs/nixos-VERSION";
```

or more simply,

```
nixpkgs.url = "nixpkgs";
```

You might be concerned that omitting the version number would make the build non-reproducible. If someone else builds the flake, could they end up with a different version of nixpkgs? No! remember that the lockfile (`flake.lock`) *uniquely* specifies all flake inputs.

Git and Mercurial repositories are the most common type of flake reference, as in the examples below.

A Git repository `git+https://github.com/NixOS/patchelf` :
A specific branch of a Git repository `git+https://github.com/NixOS/patchelf?ref=master`
A specific branch and revision of a Git repository `git+https://github.com/NixOS/patchelf?ref=mas`
A tarball `https://github.com/NixOS/patchelf/archive/master.tar.gz`

You can find more examples of flake references in the Nix Reference Manual.

Although you probably won't need to use it, there is another syntax for flake references that you might encounter. This example

```
inputs.import-cargo = {
  type = "github";
  owner = "edolstra";
  repo = "import-cargo";
};
```

is equivalent to

```
inputs.import-cargo.url = "github:edolstra/import-cargo";
```

Each of the `inputs` is fetched, evaluated and passed to the `outputs` function as a set of attributes with the same name as the corresponding input.

Outputs

This section is a function that essentially returns the recipe for building the flake.

We said above that `inputs` are passed to the `outputs`, so we need to list them as parameters. This example references the `import-cargo` dependency defined in the previous example.

```
outputs = { self, nixpkgs, import-cargo }: {
  ... outputs ...
};
```

So what actually goes in this section (where I wrote `...outputs...`)? That depends on the programming languages your software is written in, the build system you use, and more. There are Nix functions and tools that can simplify much of this, and new, easier-to-use ones are released regularly. We'll look at some of these in the next section.

A generic flake

The previous section presented a very high-level view of flakes, focusing on the basic structure. In this section, we will add a bit more detail.

Flakes are written in the Nix programming language, which is a functional language. As with most programming languages, there are many ways to achieve the same result. Below is an example you can follow when writing your own flakes. I'll explain the example in some detail.

```
{
  description = "BRIEF PACKAGE DESCRIPTION";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
    ...OTHER DEPENDENCIES... ❶
  };

  outputs = { self, nixpkgs, flake-utils, ...OTHER DEPENDENCIES... ❷ }:
    flake-utils.lib.eachDefaultSystem (system: ❸
      let
        pkgs = import nixpkgs { inherit system; };
      in
      {
        devShells = rec {
          default = pkgs.mkShell {
            packages = [ PACKAGES NEEDED FOR DEVELOPMENT SHELL ❹ ];
          };

        packages = rec {
          myPackageName = PACKAGE DEFINITION; ❺
          default = myPackageName;
        };

        apps = rec {
          myPackageName = flake-utils.lib.mkApp { drv = self.packages.${system}.myPackageName; };
          default = myPackageName; ❻
        };
      }
    );
}
```

We discussed how to specify flake inputs ❶ in the previous section, so this part of the flake should be familiar. Remember also that any dependencies in the input section should also be listed at the beginning of the outputs section ❷.

Now it's time to look at the content of the output section. If we want the package to be available for multiple systems (e.g., "x86_64-linux", "aarch64-linux", "x86_64-darwin", and "aarch64-darwin"), we need to define the output for each of those systems. Often the definitions are identical, apart from the name of the system. The eachDefaultSystem function ❸ provided by flake-utils allows us to write a single definition using a variable for the system name. The function then iterates over all default systems to generate the outputs for each one.

The `devShells` variable specifies the environment that should be available when doing development on the package. If you don't need a special development environment, you can omit this section. At ❹ you would list any tools (e.g., compilers and language-specific build tools) you want to have available in a development shell. If the compiler needs access to language-specific packages, there are Nix functions to assist with that. These functions are very language-specific, and not always well-documented. We will see examples for some languages later in the tutorial. In general, I recommend that you do a web search for "nix language", and try to find resources that were written or updated recently.

The `packages` variable defines the packages that this flake provides. The package definition ❺ depends on the programming languages your software is written in, the build system you use, and more. There are Nix functions and tools that can simplify much of this, and new, easier-to-use ones are released regularly. Again, I recommend that you do a web search for "nix language", and try to find resources that were written or updated recently.

The apps variable identifies any applications provided by the flake. In particular, it identifies the default executable ❻ that `nix run` will run if you don't specify an app.

The list below contains are a few functions that are commonly used in this section.

General-purpose The standard environment provides `mkDerivation`, which is especially useful for the typical `./configure; make; make install` scenario. It's customisable.
Python `buildPythonApplication`, `buildPythonPackage`.
Haskell `mkDerivation` (Haskell version, which is a wrapper around the standard environment version), `developPackage`, `callCabal2Nix`.

Another look at hello-flake

Now that we have a better understanding of the structure of `flake.nix`, let's have a look at the one we saw earlier, in the `hello-flake` repo. If you compare this flake definition to the colour-coded template presented in the previous section, most of it should look familiar.

```
{
  description = "a very simple and friendly flake";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
  };

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = import nixpkgs { inherit system; };
      in
      {
        packages = rec {
          hello =
            . . .
            SOME UNFAMILIAR STUFF
            . . .
          };
          default = hello;
        };

        apps = rec {
        hello = flake-utils.lib.mkApp { drv = self.packages.${system}.hello; };
          default = hello;
        };
      }
    );
}
```

This `flake.nix` doesn't have a `devShells` section, because development on the current version doesn't require anything beyond the "bare bones" linux commands. Later we will add a feature that requires additional development tools.

Now let's look at the section I labeled "`SOME UNFAMILIAR STUFF`" and see what it does.

```
        packages = rec {
          hello = pkgs.stdenv.mkDerivation rec {           # See (1) in text
            name = "hello-flake";

            src = ./.;                                     # See (2) in text

            unpackPhase = "true";

            buildPhase = ":";

            installPhase =
              ''
                mkdir -p $out/bin                          # See (3) in text
                cp $src/hello-flake $out/bin/hello-flake   # See (4) in text
                chmod +x $out/bin/hello-flake              # See (5) in text
              '';
          };
```

This flake uses `mkDerivation` (1) which is a very useful general-purpose package builder provided by the Nix standard environment. It's especially useful for the typical `./configure; make; make install` scenario, but for this flake we don't even need that.

The `name` variable is the name of the flake, as it would appear in a package listing if we were to add it to Nixpkgs or another package collection. The `src` variable (2) supplies the location of the source files, relative to `flake.nix`. When a flake is accessed for the first time, the repository contents are fetched in the form of a tarball. The `unpackPhase` variable indicates that we do want the tarball to be unpacked.

The `buildPhase` variable is a sequence of Linux commands to build the package. Typically, building a package requires compiling the source code. However, that's not required for a simple shell script. So `buildPhase` consists of a single command, `:`, which is a no-op or "do nothing" command.

The `installPhase` variable is a sequence of Linux commands that will do the actual installation. In this case, we create a directory (3) for the installation, copy the `hello-flake` script (4) there, and make the script executable (5). The environment variable `$src` refers to the source directory, which we specified earlier ❷.

Earlier we said that the build step runs in a pure environment to ensure that builds are reproducible. This means no Internet access; indeed no access to any files outside the build directory. During the build and install phases, the only commands available are those provided by the Nix standard environment and the external dependencies identified in the `inputs` section of the flake.

I've mentioned the Nix standard environment before, but I didn't explain what it is. The standard environment, or `stdenv`, refers to the functionality that is available during the build and install phases of a Nix package (or flake). It includes the commands listed below[2].

- The GNU C Compiler, configured with C and C++ support.
- GNU coreutils (contains a few dozen standard Unix commands).
- GNU findutils (contains find).
- GNU diffutils (contains diff, cmp).
- GNU sed.
- GNU grep.
- GNU awk.
- GNU tar.
- gzip, bzip2 and xz.
- GNU Make.
- Bash.

[2] For more information on the standard environment, see the Nixpkgs manual

- The patch command.
- On Linux, stdenv also includes the patchelf utility.

Only a few environment variables are available. The most interesting ones are listed below.

- `$name` is the package name.
- `$src` refers to the source directory.
- `$out` is the path to the location in the Nix store where the package will be added.
- `$system` is the system that the package is being built for.
- `$PWD` and `$TMP` both point to a temporary build directories
- `$HOME` and `$PATH` point to nonexistent directories, so the build cannot rely on them.

Modifying the flake

Let's make a simple modification to the script. This will give you an opportunity to check your understanding of flakes.

The first step is to enter a development shell.

```
$ cd ~/tutorial-practice/hello-flake
$ nix develop
```

The `flake.nix` file specifies all of the tools that are needed during development of the package. The `nix develop` command puts us in a shell with those tools. As it turns out, we didn't need any extra tools (beyond the standard environment) for development yet, but that's usually not the case. Also, we will soon need another tool.

A development environment only allows you to *develop* the package. Don't expect the package *outputs* (e.g. executables) to be available until you build them. However, our script doesn't need to be compiled, so can't we just run it?

```
$ hello-flake
bash: line 1: hello-flake: command not found
```

That worked before; why isn't it working now? Earlier we used `nix shell` to enter a *runtime* environment where `hello-flake` was available and on the `$PATH`. This time we entered a *development* environment using the `nix develop` command. Since the flake hasn't been built yet, the executable won't be on the `$PATH`. We can, however, run it by specifying the path to the script.

```
$ ./hello-flake
Hello from your flake!
```

We can also build the flake using the `nix build` command, which places the build outputs in a directory called `result`.

```
$ nix build
$ result/bin/hello-flake
Hello from your flake!
```

Rather than typing the full path to the executable, it's more convenient to use `nix run`.

```
$ nix run
Hello from your flake!
```

Here's a summary of the more common Nix commands.

command	Action
`nix develop`	Enters a *development* shell with all the required development tools (e.g. compilers and linkers) available (as specified by `flake.nix`).
`nix shell`	Enters a *runtime* shell where the flake's executables are available on the `$PATH`.
`nix build`	Builds the flake and puts the output in a directory called `result`.
`nix run`	Runs the flake's default executable, rebuilding the package first if needed. Specifically, it runs the version in the Nix store, not the version in `result`.

Now we're ready to make the flake a little more interesting. Instead of using the `echo` command in the script, we can use the Linux `cowsay` command. The `sed` command below will make the necessary changes.

```
$ sed -i 's/echo/cowsay/' hello-flake
$ cat hello-flake
#!/usr/bin/env sh

cowsay "Hello from your flake!"
```

Let's test the modified script.

```
$ ./hello-flake
./hello-flake: line 3: cowsay: command not found
```

What went wrong? Remember that we are in a *development* shell. Since `flake.nix` didn't define the `devShells` variable, the development shell only includes the Nix standard environment. In particular, the `cowsay` command is not available.

To fix the problem, we can modify `flake.nix`. We don't need to add `cowsay` to the `inputs` section because it's included in `nixpkgs`, which is already an input. However, we do need to indicate that we want it available in a develoment shell. Add the following lines before the `packages = rec {` line.

```
        devShells = rec {
          default = pkgs.mkShell {
            packages = [ pkgs.cowsay ];
          };
        };
```

Here is a "diff" showing the changes in `flake.nix`.

```
$ git diff flake.nix
diff --git a/flake.nix b/flake.nix
index c1b7807..42862f1 100644
--- a/flake.nix
+++ b/flake.nix
@@ -12,6 +12,12 @@
          pkgs = import nixpkgs { inherit system; };
        in
        {
+         devShells = rec {
+           default = pkgs.mkShell {
+             packages = [ pkgs.cowsay ];
+           };
+         };
+
          packages = rec {
            hello = pkgs.stdenv.mkDerivation rec {
              name = "hello-flake";
```

We restart the development shell and see that the `cowsay` command is now available and the script works. Because we've updated source files but haven't `git commited` the new version, we get a warning message about it being "dirty". It's just a warning, though; the script runs correctly.

```
$ nix develop
warning: Git tree '/home/amy/tutorial-practice/hello-flake' is dirty
warning: Git tree '/home/amy/tutorial-practice/hello-flake' is dirty
$ which cowsay # is it available now?
/nix/store/gfi27h4y5n4aralcxrc0377p8mjb1cvb-cowsay-3.7.0/bin/cowsay
$ ./hello-flake
 _____________________
< Hello from your flake! >
 ---------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Alternatively, we could use `nix run`.

```
$ nix run
warning: Git tree '/home/amy/tutorial-practice/hello-flake' is dirty
 _____________________
< Hello from your flake! >
 ---------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Note, however, that `nix run` rebuilt the package in the Nix store and ran *that*. It did not alter the copy in the `result` directory, as we'll see next.

```
$ cat result/bin/hello-flake
#!/nix/store/zlf0f88vj30sc7567b80l52d19pbdmy2-bash-5.2-p15/bin/sh

echo "Hello from your flake!"
```

If we want to update the version in `result`, we need `nix build` again.

```
$ nix build
warning: Git tree '/home/amy/tutorial-practice/hello-flake' is dirty
warning: Git tree '/home/amy/tutorial-practice/hello-flake' is dirty
$ cat result/bin/hello-flake
#!/nix/store/zlf0f88vj30sc7567b80l52d19pbdmy2-bash-5.2-p15/bin/sh

cowsay "Hello from your flake!"
```

Let's `git commit` the changes and verify that the warning goes away. We don't need to `git push` the changes until we're ready to share them.

```
$ git commit hello-flake flake.nix -m 'added bovine feature'
[main b3bbf77] added bovine feature
 2 files changed, 7 insertions(+), 1 deletion(-)
$ nix run
 _________________________
< Hello from your flake! >
 -------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Development workflows

If you're getting confused about when to use the different commands, it's because there's more than one way to use Nix. I tend to think of it as two different development workflows.

My usual, *high-level workflow* is quite simple.

1. `nix run` to re-build (if necessary) and run the executable.
2. Fix any problems in `flake.nix` or the source code.
3. Repeat until the package works properly.

In the high-level workflow, I don't use a development shell because I don't need to directly invoke development tools such as compilers and linkers. Nix invokes them for me according to the output definition in `flake.nix`.

Occasionally I want to work at a lower level, and invoke compiler, linkers, etc. directly. Perhaps want to work on one component without rebuilding the entire package. Or perhaps I'm confused by some error message, so I want to temporarily bypass Nix and "talk" directly to the compiler. In this case I temporarily switch to a *low-level workflow*.

1. `nix develop` to enter a development shell with any development tools I need (e.g. compilers, linkers, documentation generators).
2. Directly invoke tools such as compilers.
3. Fix any problems in `flake.nix` or the source code.

4. Directly invoke the executable. Note that the location of the executable depends on the development tools – It probably isn't `result`!
5. Repeat until the package works properly.

I generally only use `nix build` if I just want to build the package but not execute anything (perhaps it's just a library).

This all seems like a hassle!

It is a bit annoying to modify `flake.nix` and ether rebuild or reload the development environment every time you need another tool. However, this Nix way of doing things ensures that all of your dependencies, down to the exact versions, are captured in `flake.lock`, and that anyone else will be able to reproduce the development environment.

A new flake from scratch (Python)

At last we are ready to create a flake from scratch! Start with an empty directory and create a git repository.

```
$ cd ~/tutorial-practice
$ mkdir hello-python
$ cd hello-python
$ git init
Initialized empty Git repository in /home/amy/tutorial-practice/hello-python/.git/
```

Next, we'll create a simple Python program.

```
$ cat hello.py
#!/usr/bin/env python

def main():
    print("Hello from inside a Python program built with a Nix flake!")

if __name__ == "__main__":
    main()
```

Before we pachage the program, let's verify that it runs. We're going to need Python. By now you've probably figured out that we can write a `flake.nix` and define a development shell that includes Python. We'll do that shortly, but first I want to show you a handy shortcut. We can lauch a *temporary* shell with any Nix packages we want. This is convenient when you just want to try out some new software and you're not sure if you'll use it again. It's also convenient when you're not ready to write `flake.nix` (perhaps you're not sure what tools and packages you need), and you want to experiment a bit first.

The command to enter a temporary shell is

`nix-shell -p` *packages*

If there are multiple packages, they should be separated by spaces. Note that the command used here is `nix-shell` with a hyphen, not `nix shell` with a space; those are two different commands. In fact there are hyphenated and non-hyphenated versions of many Nix commands, and yes, it's confusing. The non-hyphenated commands were introduced when support for flakes was added to Nix. I predict that eventually all hyphenated commands will be replaced with non-hyphenated versions. Until then, a useful rule of thumb is that non-hyphenated commands are for for working directly with flakes; hyphenated commands are for everything else.

Let's enter a shell with Python so we can test the program.

```
$ nix-shell -p python3
$ python hello.py
Hello from inside a Python program built with a Nix flake!
```

Next, create a Python script to build the package. We'll use Python's setuptools, but you can use other build tools. For more information on setuptools, see the Python Packaging User Guide, especially the section on setup args.

```
$ cat setup.py
#!/usr/bin/env python

from setuptools import setup

setup(
    name='hello-flake-python',
    version='0.1.0',
    py_modules=['hello'],
    entry_points={
        'console_scripts': ['hello-flake-python = hello:main']
    },
)
```

We won't write `flake.nix` just yet. First we'll try building the package manually.

```
$ python -m build
/nix/store/iw1vmh509hcbby8dbpsaanbri4zsq7dj-python3-3.10.10/bin/python: No module named bu
```

The missing module error happens because we don't have `build` available in the temporary shell. We can fix that by adding "build" to the temporary shell. When you need support for both a language and some of its packages, it's best to use one of the Nix functions that are specific to the programming language and build system. For Python, we can use the `withPackages` function.

```
$ nix-shell -p "python3.withPackages (ps: with ps; [ build ])"
```

Note that we're now inside a temporary shell inside the previous temporary shell! To get back to the original shell, we have to `exit` twice. Alternatively, we could have done `exit` followed by the `nix-shell` command.

```
$ python -m build
```

After a lot of output messages, the build succeeds.

Now we should write `flake.nix`. We already know how to write most of the flake from the examples we did earlier. The two parts that will be different are the development shell and the package builder.

Let's start with the development shell. It seems logical to write something like the following.

```
        devShells = rec {
          default = pkgs.mkShell {
            packages = [ (python.withPackages (ps: with ps; [ build ])) ];
          };
        };
```

Note that we need the parentheses to prevent `python.withPackages` and the argument from being processed as two separate tokens. Suppose we wanted to work with `virtualenv` and `pip` instead of `build`. We could write something like the following.

```
          devShells = rec {
            default = pkgs.mkShell {
              packages = [
                # Python plus helper tools
                (python.withPackages (ps: with ps; [
                  virtualenv # Virtualenv
                  pip # The pip installer
                ]))
              ];
            };
          };
```

For the package builder, we can use the `buildPythonApplication` function.

```
          packages = rec {
            hello = python.pkgs.buildPythonApplication {
              name = "hello-flake-python";
              buildInputs = with python.pkgs; [ pip ];
              src = ./.;
            };
            default = hello;
          };
```

If you put all the pieces together, your `flake.nix` should look something like this.

```
$ cat flake.nix
{
  # See https://github.com/mhwombat/nix-for-numbskulls/blob/main/flakes.md
  # for a brief overview of what each section in a flake should or can contain.

  description = "a very simple and friendly flake written in Python";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
  };

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = import nixpkgs { inherit system; };
        python = pkgs.python3;
      in
      {
        devShells = rec {
          default = pkgs.mkShell {
            packages = [
              # Python plus helper tools
              (python.withPackages (ps: with ps; [
                virtualenv # Virtualenv
                pip # The pip installer
              ]))
            ];
          };
        };

        packages = rec {
          hello = python.pkgs.buildPythonApplication {
            name = "hello-flake-python";

            buildInputs = with python.pkgs; [ pip ];

            src = ./.;
          };
          default = hello;
        };

        apps = rec {
        hello = flake-utils.lib.mkApp { drv = self.packages.${system}.hello; };
          default = hello;
        };
      }
    );
}
```

Let’s try out the new flake.

```
$ nix run
warning: Git tree '/home/amy/tutorial-practice/hello-python' is dirty
error: getting status of '/nix/store/0ccnxa25whszw7mgbgyzdm4nqc0zwnm8-source/flake.nix': N
```

Why can't it find `flake.nix`? Nix flakes only "see" files that are part of the repository. We need to add all of the important files to the repo before building or running the flake.

```
$ git add flake.nix setup.py hello.py
$ nix run
warning: Git tree '/home/amy/tutorial-practice/hello-python' is dirty
warning: creating lock file '/home/amy/tutorial-practice/hello-python/flake.lock'
warning: Git tree '/home/amy/tutorial-practice/hello-python' is dirty
Hello from inside a Python program built with a Nix flake!
```

We'd like to share this package with others, but first we should do some cleanup. When the package was built (automatically by the `nix run` command), it created a `flake.lock` file. We need to add this to the repo, and commit all important files.

```
$ git add flake.lock
$ git commit -a -m 'initial commit'
[master (root-commit) 028c151] initial commit
 4 files changed, 127 insertions(+)
 create mode 100644 flake.lock
 create mode 100644 flake.nix
 create mode 100644 hello.py
 create mode 100644 setup.py
```

You can test that your package is properly configured by going to another directory and running it from there.

```
$ cd ~
$ nix run ~/tutorial-practice/hello-python
Hello from inside a Python program built with a Nix flake!
```

If you move the project to a public repo, anyone can run it. Recall from the beginning of the tutorial that you were able to run `hello-flake` directly from my repo with the following command.

```
nix run "git+https://codeberg.org/mhwombat/hello-flake"
```

Modify the URL accordingly and invite someone else to run your new Python flake.