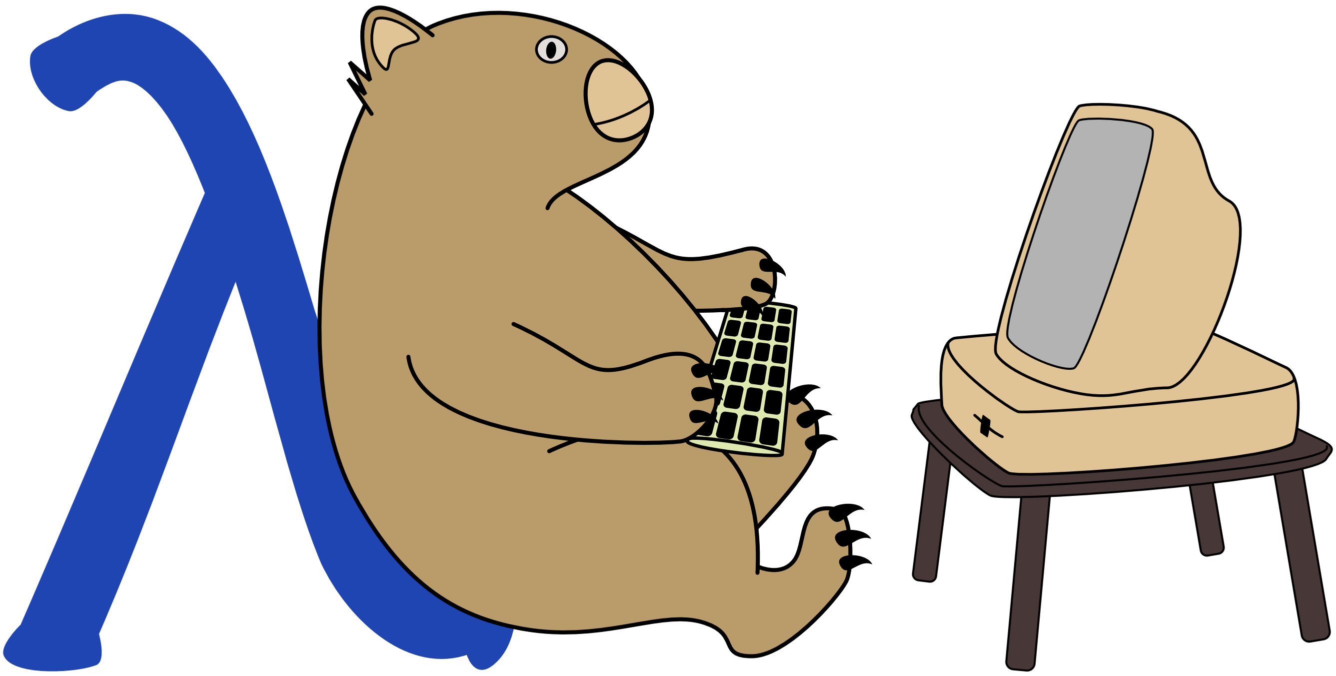
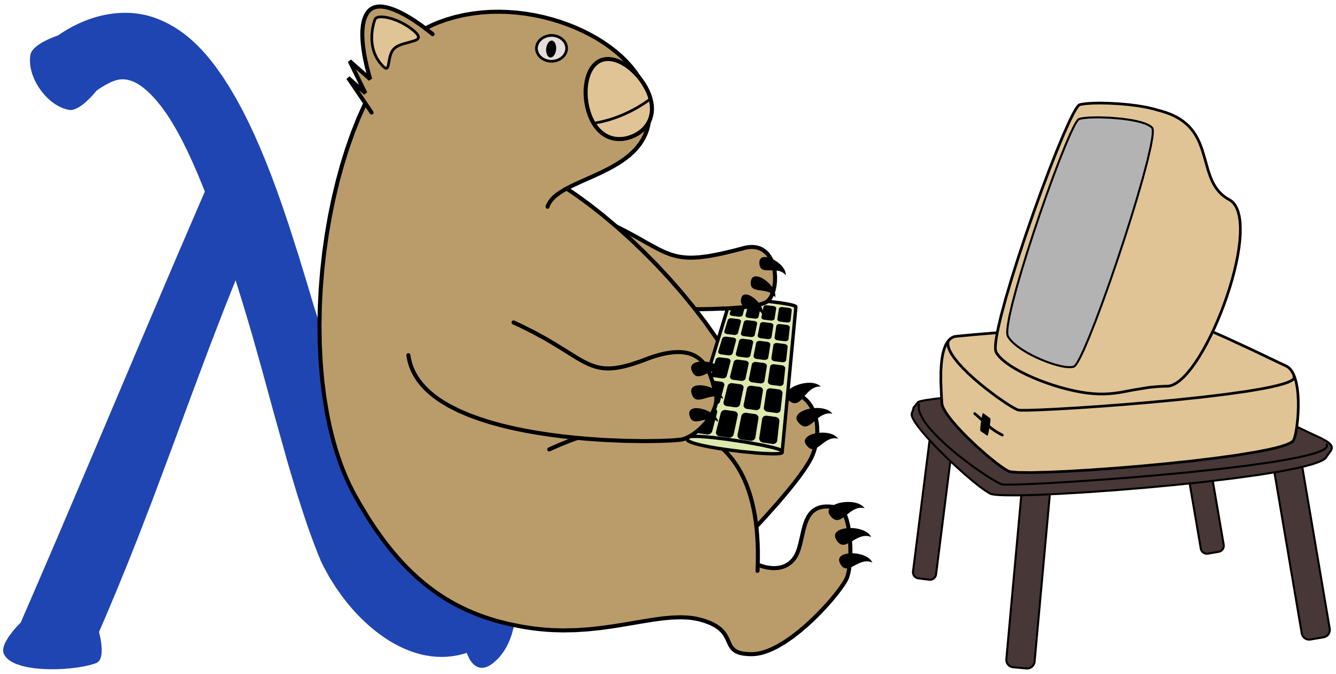


Wombat’s Book of Nix

Amy de Buitléir

Table of Contents



This book is available online and as a downloadable PDF.

Chapter 1. Introduction

1.1. Why Nix?

If you've opened this PDF, you already have your own motivation for learning Nix. Here's how it helps me. As a researcher, I tend to work on a series of short-term projects, mostly demos and prototypes. For each one, I typically develop some software using a compiler, often with some open source libraries. Often I use other tools to analyse data or generate documentation, for example.

Problems would arise when handing off the project to colleagues; they would report errors when trying to build or run the project. Belatedly I would realise that my code relies on a library that they need to install. Or perhaps they had installed the library, but the version they're using is incompatible.

Using containers helped with the problem. However, I didn't want to *develop* in a container. I did all my development in my nice, familiar, environment with my custom aliases and shell prompt. and *then* I containerised the software. This added step was annoying for me, and if my colleague wanted to do some additional development, they would probably extract all of the source code from the container first anyway. Containers are great, but this isn't the ideal use case for them.

Nix allows me to work in my custom environment, but forces me to specify any dependencies. It automatically tracks the version of each dependency so that it can replicate the environment wherever and whenever it's needed.

1.2. Why *flakes*?

Flakes are labeled as an experimental feature, so it might seem safer to avoid them. However, they have been in use for years, and there is widespread adoption, so the aren't going away any time soon. Flakes are easier to understand, and offer more features than the traditional Nix approach. After weighing the pros and cons, I feel it's better to learn and use flakes; and this seems to be the general consensus.

1.3. Prerequisites

To follow along with the examples in this book, you will need access to a computer or (virtual machine) with Nix (or NixOS) installed and *flakes enabled*.

I recommend the installer from zero-to-nix.com. This installer automatically enables flakes.

More documentation (and another installer) available at nixos.org.

To *enable flakes on an existing Nix or NixOS installation*, see the instructions in the NixOS wiki.

> There are hyphenated and unhyphenated versions of many Nix commands. For example, `nix-shell` and `nix shell` are two different commands. Don't confuse them!
>
> Generally speaking, the unhyphenated versions are for working directly with

> flakes, while the hyphenated versions are for everything else.

1.4. See an error? Or want more?

If notice an error, or you're interested in an area that isn't covered in this book, feel free to open an issue.

Chapter 2. The Nix language

2.1. Introducing the Nix language

Nix and NixOS use a functional programming language called *Nix* to specify how to build and install software, and how to configure system, user, and project-specific environments. (Yes, “Nix” is the name of both the package manager and the language it uses.)

Nix is a *functional* language. In a *procedural* language such as C or Java, the focus is on writing a series of *steps* (statements) to achieve a desired result. By contrast, in a functional language the focus is on *defining* the desired result.

In the case of Nix, the desired result is usually a *derivation*: a software package that has been built and is ready for use. The Nix language has been designed for that purpose, and thus has some features you don’t typically find in general-purpose languages.

2.2. Data types

2.2.1. Strings

Strings are enclosed by double quotes (`"`), or *two* single quotes (`'`).

```
"Hello, world!"
```

```
''This string contains "double quotes"''
```

They can span multiple lines.

```
''Old pond
A frog jumps in
The sound of water
          -- Basho''
```

2.2.2. Integers

```
7
256
```

2.2.3. Floating point numbers

```
3.14
```

```
6.022e23
```

2.2.4. Boolean

The Boolean values in Nix are `true` and `false`.

2.2.5. Paths

File paths are play an important role in building software, so Nix has a special data type for them. Paths may be absolute (e.g. `/bin/sh`) or relative (e.g. `./data/file1.csv`). Note that paths are not enclosed in quotation marks; they are not strings!

Enclosing a path in angle brackets, e.g. <nixpkgs> causes the directories listed in the environment variable NIX_PATH to be searched for the given file or directory name. These are called *lookup paths*.

2.2.6. Lists

List elements are enclosed in square brackets and separated by spaces (not commas). The elements need not be of the same type.

```
[ "apple" 123 ./build.sh false ]
```

Lists can be empty.

```
[]
```

List elements can be of any type, and can even be lists themselves.

```
[ [ 1 2 ] [ 3 4 ] ]
```

2.2.7. Attribute sets

Attribute sets associate keys with values. They are enclosed in curly brackets, and the associations are terminated by semi-colons. Note that the final semi-colon before the closing bracket is required.

```
{ name = "Professor Paws"; age = 10; species = "cat"; }
```

The keys of an attribute set must be strings. When the key is not a valid identifier, it must be enclosed in quotation marks.

```
{ abc = true; "123" = false; }
```

Attribute sets can be empty.

```
{}
```

Values of attribute sets can be of any type, and can even be attribute sets themselves.

```
{ name = { first = "Professor"; last = "Paws"; }; age = 10; species = "cat"; }
```

In some Nix documentation, and in many articles about Nix, attribute sets are simply called "sets".

2.2.8. Functions

We'll learn how to write functions later in this chapter. For now, note that functions are "first-class values", meaning that they can be treated like any other data type. For example, a function can be assigned to a variable, appear as an element in a list, or be associated with a key in an attribute set.

```
[ "apple" 123 ./build.sh false (x: x*x) ]
{ name = "Professor Paws"; age = 10; species = "cat"; formula = (x: x*2); }
```

2.3. Stop reading this chapter!

When I first began using Nix, it seemed logical to start by learning the Nix language. However, after following an in-depth tutorial, I found that I didn't know how to do anything useful with the language! It wasn't until later that I understood what I was missing: a guide to the most useful library functions.

When working with Nix or NixOS, it's very rare that you'll want to write something from scratch. Instead, you'll use one of the many library functions that make things easier and shield you from the underlying complexity. Many of these functions are language-specific, and the documentation for them may be inadequate. Often the easiest (or only) way to learn to use them is to find an example that does something similar to what you want, and then modify the function parameters to suit your needs.

At this point you've learned enough of the Nix language to do the majority of common Nix tasks. So when I say "Stop reading this chapter!", I'm only half-joking. Instead I suggest that you *skim* the rest of this chapter, paying special attention to anything marked with . Then move on to the following chapters where you will learn how to develop and package software using Nix. Afterward, come back to this chapter and read it in more detail.

While writing this book, I anticipated that readers would want to skip around, alternating between pure learning and learning-by-doing. I've tried to structure the book to support that; sometimes repeating information from earlier chapters that you might have skipped.

2.4. The Nix REPL

The Nix REPL [1] is an interactive environment for evaluating and debugging Nix code. It's also a good place to begin learning Nix. Enter it using the command `nix repl`. Within the REPL, type `:?` to see a list of available commands.

```
$# echo "$ nix repl"
Welcome to Nix 2.18.1. Type :? for help.

nix-repl> :?
The following commands are available:

  <expr>                       Evaluate and print expression
  <x> = <expr>                 Bind expression to variable
  :a, :add <expr>              Add attributes from resulting set to scope
  :b <expr>                    Build a derivation
  :bl <expr>                   Build a derivation, creating GC roots in the
                               working directory
  :e, :edit <expr>             Open package or function in $EDITOR
  :i <expr>                    Build derivation, then install result into
                               current profile
  :l, :load <path>             Load Nix expression and add it to scope
  :lf, :load-flake <ref>       Load Nix flake and add it to scope
  :p, :print <expr>            Evaluate and print expression recursively
  :q, :quit                    Exit nix-repl
  :r, :reload                  Reload all files
  :sh <expr>                   Build dependencies of derivation, then start
                               nix-shell
  :t <expr>                    Describe result of evaluation
  :u <expr>                    Build derivation, then start nix-shell
  :doc <expr>                  Show documentation of a builtin function
  :log <expr>                  Show logs for a derivation
  :te, :trace-enable [bool]    Enable, disable or toggle showing traces for
                               errors
  :?, :help                    Brings up this help menu
```

A command that is useful to beginners is `:t`, which tells you the type of an expression.

Note that the command to exit the REPL is `:q` (or `:quit` if you prefer).

2.5. Variables

You can declare variables in Nix and assign values to them.

```
nix-repl> a = 7

nix-repl> b = 3
```

```
nix-repl> a - b
4
```

As mentioned previously, omitting the spaces around operators can have unexpected results. As Nix allows hyphens (`-`) in variable names, `a-b` is interpreted as the name of a variable in the following example.

```
nix-repl> a-b
error: undefined variable 'a-b'

       at «string»:1:1:

            1| a-b
             | ^
```

In Nix, values are *immutable*; once you assign a value to a variable, you cannot change it. You can, however, create a new variable with the same name, but in a different scope. Don't worry if you don't completely understand the previous sentence; it should become clearer when we discuss functions (Section Functions), let expressions (Section Let expressions), and `with` expressions (Section With expressions).

```
nix-repl> x = 1

nix-repl> x
1

nix-repl> x = x + 1

nix-repl> x
2
```

2.6. Numeric operations

The usual arithmetic operators are provided.

```
nix-repl> 1 + 2   # addition
3

nix-repl> 5 - 3   # subtraction
2

nix-repl> 3 * 4   # multiplication
12

nix-repl> 6 / 2   # division
```

```
3

nix-repl> -1      # negation
-1
```

The spaces before and after operators aren't always required. However, you can get unexpected results when you omit them, as shown below.

```
nix-repl> 6/2
/home/amy/codeberg/nix-book/6/2
```

What happened? Let's use the `:t` command to find out the type of the expression.

```
nix-repl> :t 6/2
```

If an expression can be interpreted as a path, Nix will do so. This is useful, because paths are *far* more commonly used in Nix expressions that arithmetic operators. In this case, Nix interpreted `6/2` as a relative path from the current directory, which in the above example was `/home/amy/codeberg/nix-book`.

Adding a space after the `/` operator produces the expected result.

```
nix-repl> 6/ 2
3
```

To avoid surprises and improve readability, I prefer to use spaces before and after all operators.

Numbers without a decimal point are assumed to be integers. To ensure that a number is interpreted as a floating-point value, add a decimal point.

```
nix-repl> :t 5
an integer

nix-repl> :t 5.0
a float

nix-repl> :t 5.
a float
```

In the example below, the first expression results in integer division (rounding down), while the second produces a floating-point result.

```
nix-repl> 5 / 3
```

```
1

nix-repl> 5.0 / 3
1.66667
```

2.7. String operations

String concatenation uses the `+` operator.

```
nix-repl> "Hello, " + "world!"
"Hello, world!"
```

You can use the `${variable}` syntax to insert the value of a variable within a string.

```
nix-repl> name = "Wombat"

nix-repl> "Hi, I'm ${name}."
"Hi, I'm Wombat."
```

You cannot mix numbers and strings. Nix does provide functions for converting between types; we'll see these later. Earlier we set `a = 7`, so the following expression fails.

```
nix-repl> "My favourite number is ${a}."
error:
       … while evaluating a path segment

         at «string»:1:25:

            1| "My favourite number is ${a}."
             |                         ^

       error: cannot coerce an integer to a string
```

2.8. Boolean operations

The usual boolean operators are available. Recall that earlier we set `a = 7` and `b = 3`.

```
nix-repl> a == 7                    # equality test
true

nix-repl> b != 3                    # inequality
false
```

```
nix-repl> a > 12                      # greater than
false

nix-repl> b >= 2                      # greater than or equal
true

nix-repl> a < b                       # less than
false

nix-repl> b <= a                      # less than or equal
true

nix-repl> !true                       # logical negation
false

nix-repl> (3 * a == 21) && (a > b)    # logical AND
true

nix-repl> (b > a) || (b > 10)         # logical OR
false
```

One operator that might be unfamiliar to you is *logical implication*, which uses the symbol `->`. The expression `u -> v` is equivalent to `!u || v`.

```
nix-repl> u = false

nix-repl> v = true

nix-repl> u -> v
true

nix-repl> v -> u
false
```

2.9. Path operations

Any expression that contains a forward slash (`/`) *not* followed by a space is interpreted as a path. To refer to a file or directory relative to the current directory, prefix it with `./`. You can specify the current directory as `./.`.

```
nix-repl> ./file.txt
/home/amy/codeberg/nix-book/file.txt

nix-repl> ./.
/home/amy/codeberg/nix-book
```

Paths can be concatenated with each other to produce a new path.

```
nix-repl> /home/wombat + /bin/sh
/home/wombat/bin/sh

nix-repl> :t /home/wombat + /bin/sh
a path
```

Relative paths become absolute when they are parsed, which occurs before concatenation. This is why the result in the example below is not `/home/amy/file.txt`.

```
nix-repl> /home/wombat + ./file.txt
/home/wombat/home/amy/codeberg/nix-book/file.txt
```

A path can be concatenated with a string to produce a new path.

```
nix-repl> homePath + "/file.txt"
/home/wombat/file.txt

nix-repl> :t homePath + "/myfile.txt"
a path
```

The Nix reference manual says that the string must not "have a string context" that refers to a store path. String contexts are beyond the scope of this book; for more information see https://nixos.org/manual/nix/stable/language/operators#path-concatenation.

Strings can be concatenated with paths, but with a side-effect that may surprise you: if the path exists, the file is copied to the Nix store! The result is a string, not a path.

In the example below, the file `file.txt` is copied to `/nix/store/gp8ba25gpwvbqizqfr58jr014gmv1hd8-file.txt` (not `/home/wombat/nix/store/gp8ba25gpwvbqizqfr58jr014gmv1hd8-file.txt`).

```
nix-repl> "/home/wombat" + ./file.txt
"/home/wombat/nix/store/gp8ba25gpwvbqizqfr58jr014gmv1hd8-file.txt"
```

The path must exist.

```
nix-repl> "/home/wombat" + ./no-such-file.txt
error (ignored): error: end of string reached
error: getting status of '/home/amy/codeberg/nix-book/no-such-file.txt': No such file
or directory
```

2.10. List operations

Lists can be concatenated using the ++ operator.

```
nix-repl> [ 1 2 3 ] ++ [ "apple" "banana" ]
[ 1 2 3 "apple" "banana" ]
```

2.11. Attribute set operations

An ordinary attribute set cannot refer to its own elements. To do this, you need a *recursive* attribute set.

```
nix-repl> { x = 3; y = 4*x; }
error: undefined variable 'x'

       at «string»:1:16:

            1| { x = 3; y = 4*x; }
             |                ^

nix-repl> rec { x = 3; y = 4*x; }
{ x = 3; y = 12; }
```

The . operator selects an attribute from a set.

```
nix-repl> animal = { name = { first = "Professor"; last = "Paws"; }; age = 10; species
= "cat"; }

nix-repl> animal . age
10

nix-repl> animal . name . first
"Professor"
```

We can use the ? operator to find out if a set has a particular attribute.

```
nix-repl> animal ? species
true

nix-repl> animal ? bicycle
false
```

We can use the // operator to modify an attribute set. Recall that Nix values are immutable, so the result is a new value (the original is not modified).

```
nix-repl> animal // { species = "tiger"; }
{ age = 10; name = { ... }; species = "tiger"; }
```

2.12. Functions

2.13. Built-in functions

For a complete list of built-in functions, see https://nixos.org/manual/nix/stable/language/builtins

2.14. If expressions

The conditional construct in Nix is an *expression*, not a *statement*. Since expressions must have values in all cases, you must specify both the `then` and the `else` branch.

```
nix-repl> a = 7

nix-repl> b = 3

nix-repl> if a > b then "yes" else "no"
"yes"
```

2.15. Let expressions

A `let` expression defines a value with a local scope.

```
nix-repl> let x = 3; in x*x
9

nix-repl> let x = 3; y = 2; in x*x + y
11
```

You can also nest `let` expressions. The previous expression is equivalent to the following.

```
nix-repl> let x = 3; in let y = 2; in x*x + y
11
```

A variable defined inside a `let` expression will "shadow" an outer variable with the same name.

```
nix-repl> x = 100

nix-repl> let x = 3; in x*x
```

```
9

nix-repl> let x = 3; in let x = 7; in x+1
8
```

A variable in a let expression can refer to another variable in the expression. This is similar to how recursive attribute sets work.

```
nix-repl> let x = 3; y = x + 1; in x*y
12
```

2.16. With expressions

A `with` expression is somewhat similar to a `let` expression, but it brings all of the associations in an attribute set into scope.

```
nix-repl> point = { x1 = 3; x2 = 2; }

nix-repl> with point; x1*x1 + x2
11
```

Unlike a `let` expression, a variable defined inside a `with` expression will *not* "shadow" an outer variable with the same name.

```
nix-repl> name = "Amy"

nix-repl> animal = { name = "Professor Paws"; age = 10; species =
"cat"; }

nix-repl> with animal; "Hello, " + name
"Hello, Amy"
```

However, you can refer to the variable in the inner scope using the attribute selection operator (`.`).

```
nix-repl> with animal; "Hello, " + animal.name
"Hello, Professor Paws"
```

[1] REPL is an acronym for (Read-Eval-Print-Loop).

Chapter 3. 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 flake, Nix has to fetch and build it, which may take time. 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 on 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/qskl8ajlgnl654fhgsmv74yv8x9r3kzg-hello-flake/bin/hello-flake
```

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

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

However, we can still run 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
```

Chapter 4. The hello-flake repo

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

```
$ 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 executable we ran earlier. This particular executable is just a shell script, so we can view it. It's an extremly simple script with just two lines.

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`.

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";

  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 the Nix language, introduced in an earlier chapter. However, you don't really need to know Nix to follow this example. 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 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.

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.

Chapter 5. 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
}
```

5.1. Description

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 when I began using flakes.

5.2. 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 of a flake reference, 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 *version* 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";
```

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 revision of a Git repository
: `git+https://github.com/NixOS/patchelf?ref=master&rev=f34751b88bd07d7f44f5cd3200fb4122bf916c7e`

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.

5.3. 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 }: {
  definitions for outputs
};
```

So what actually goes in the highlighted section? 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.

Chapter 6. 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; };
        python = pkgs.python3;
      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;
        };
      }
    );
}
```

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 that `nix run` will run if you don't specify an app.

Below is a list of some 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`.

Chapter 7. 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.

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 =
            . . .
            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 { ❶
            name = "hello-flake";
```

```
            src = ./.; ❷

            unpackPhase = "true";

            buildPhase = ":";

            installPhase =
              ''
                mkdir -p $out/bin ❸
                cp $src/hello-flake $out/bin/hello-flake ❹
                chmod +x $out/bin/hello-flake ❺
              '';
          };
```

This flake uses `mkDerivation` ❶ 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 ❷ 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 ❸ for the installation, copy the `hello-flake` script there ❹, and make the script executable ❺. 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[1].

- 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.
- 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.

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

Chapter 8. Modifying the flake

8.1. The Nix development shell

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.

```
$ 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 16: 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`.

8.2. Introducing a dependency

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. Here's the `hello-flake` file, with the modified line highlighted.

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 also want it to be available in a develoment shell. The highlighted modifications below will accomplish that.

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";

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

        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;
        };
      }
    );
}
```

Now we restart the development shell and see that the `cowsay` command is available and the script works. Because we've updated source files but haven't `git commit`ed 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/codeberg/nix-book/source/modify-hello-flake/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/codeberg/nix-book/source/modify-hello-flake/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/codeberg/nix-book/source/modify-hello-flake/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 eb4d08c] added bovine feature
 2 files changed, 7 insertions(+), 1 deletion(-)
$ nix run
 ________________________
< Hello from your flake! >
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

8.3. 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 work directly with 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).

8.4. 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.

Chapter 9. A new flake from scratch

At last we are ready to create a flake from scratch! The sections in this chapter are very similar; read the one for your language of choice. If you're interested in a language that I haven't covered, feel free to suggest it by creating an issue.

9.1. Haskell

Start with an empty directory and create a git repository.

```
$ mkdir hello-haskell
$ cd hello-haskell
$ git init
Initialized empty Git repository in /home/amy/codeberg/nix-book/source/new-
flake/haskell-flake/hello-haskell/.git/
```

9.1.1. A simple Haskell program

Next, we'll create a simple Haskell program.

Main.hs

```haskell
import Network.HostName

main :: IO ()
main = do
  putStrLn "Hello from Haskell inside a Nix flake!"
  h <- getHostName
  putStrLn $ "Your hostname is: " ++ h
```

9.1.2. (Optional) Testing before packaging

Before we package the program, let's verify that it runs. We're going to need a Haskell compiler. By now you've probably figured out that we can write a `flake.nix` and define a development shell that includes Haskell. 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.

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.

Some unsuitable shells

In this section, we will try commands that fail in subtle ways. Examining these failures will give you a much better understanding of Haskell development with Nix, and help you avoid (or at least diagnose) similar problems in future. If you're impatient, you can skip to the next section to see the right way to do it. You can come back to this section later to learn more.

Let's enter a shell with the Glasgow Haskell Compiler ("ghc") and try to run the program.

```
$ nix-shell -p ghc
$ runghc Main.hs

Main.hs:1:1: error:
    Could not find module Network.HostName
    Use -v (or `:set -v` in ghci) to see a list of the files searched for.
  |
1 | import Network.HostName
  | ^^^^^^^^^^^^^^^^^^^^^^^
```

The error message tells us that we need the module `Network.HostName`. That module is provided by the Haskell package called `hostname`. Let's exit that shell and try again, this time adding the `hostname` package.

```
$ exit
$ nix-shell -p "[ghc hostname]"
$ runghc Main.hs

Main.hs:1:1: error:
    Could not find module Network.HostName
    Use -v (or `:set -v` in ghci) to see a list of the files searched for.
  |
1 | import Network.HostName
  | ^^^^^^^^^^^^^^^^^^^^^^^
```

That reason that failed is that we asked for the wrong package. The Nix package `hostname` isn't the Haskell package we wanted, it's a different package entirely (an alias for `hostname-net-tools`.) The package we want is in the *package set* called `haskellPackages`, so we can refer to it as `haskellPackages.hostname`.

Let’s try that again, with the correct package.

```
$ exit
$ nix-shell -p "[ghc haskellPackages.hostname]"
$ runghc Main.hs

Main.hs:1:1: error:
    Could not find module ‘Network.HostName’
    Use -v (or `:set -v` in ghci) to see a list of the files searched for.
  |
1 | import Network.HostName
  | ^^^^^^^^^^^^^^^^^^^^^^^
```

Now what’s wrong? The syntax we used in the `nix-shell` command above is fine, but it doesn’t make the package *available to GHC*!

A suitable shell for a quick test

Consider the Haskell "pandoc" package, which provides both an executable (the Nix package `pandoc`) and a library (the Nix package `haskellPackages.pandoc`). There are several different shells we could create involving both Pandoc and GHC, and it’s important to understand the differences between them.

`nix-shell -p "[ghc pandoc]"`	Makes the Pandoc *executable* available at the command line, but the *library* won’t be visible to GHC.
`nix-shell -p "haskellPackages.ghcWithPackages (pkgs: with pkgs; [ pandoc ])"`	Makes the Pandoc *library* visible to GHC, but we won’t be able to run the *executable*.
`nix-shell -p "[pandoc (haskellPackages.ghcWithPackages (pkgs: with pkgs; [ pandoc ]))]"`	Makes the Pandoc *executable* available at the command line, and the *library* visible to GHC.

Now we can create a shell that can run the program.

```
$ nix-shell -p "haskellPackages.ghcWithPackages (pkgs: with pkgs; [ hostname ])"
$ runghc Main.hs
Hello from Haskell inside a Nix flake!
Your hostname is: wombat11k
```

Success! Now we know the program works.

9.1.3. The cabal file

It’s time to write a Cabal file for this program. This is just an ordinary Cabal file; we don’t need to do anything special for Nix.

hello-flake-haskell.cabal

```
cabal-version:   3.0
name:            hello-flake-haskell
version:         1.0.0
synopsis:        A simple demonstration using a Nix flake to package a Haskell program that prints a greeting.
description:
  For more information and a tutorial on how to use this package,
  please see the README at <https://codeberg.org/mhwombat/hello-flake-haskell#readme>.
homepage:        https://codeberg.org/mhwombat/hello-flake-haskell
bug-reports:     https://codeberg.org/mhwombat/hello-flake-haskell/issues
license:         GPL-3.0-only
license-file:    LICENSE
author:          Amy de Buitléir
maintainer:      amy@nualeargais.ie
copyright:       (c) 2023 Amy de Buitléir
category:        Text
build-type:      Simple

executable hello-flake-haskell
  main-is:          Main.hs
  build-depends:
    base,
    hostname
-- NOTE: Best practice is to specify version constraints for the packages we depend on.
-- However, I'm confident that this package will only be used as a Nix flake.
-- Nix will automatically ensure that anyone running this program is using the
-- same library versions that I used to build it.
```

9.1.4. (Optional) Building and running with cabal-install

At this point, I would normally write `flake.nix` and use Nix to build the program. I'll cover that in the next section. However, it's useful to know how to build the package manually in a Nix envronment, without using a Nix flake. When you're new to Nix, this can help you differentiate between problems in your flake definition and problems in your Cabal file.

```
$ cabal build
sh: line 35: cabal: command not found
```

Aha! We need `cabal-install` in our shell. Rather than launch another shell-within-a-shell, let's exit create a new one.

```
$ exit
$ nix-shell -p "[ cabal-install (haskellPackages.ghcWithPackages (pkgs: with pkgs; [ hostname ]))]"
```

```
$ cabal build
Warning: The package list for 'hackage.haskell.org' is 21 days old.
Run 'cabal update' to get the latest list of available packages.
Resolving dependencies...
Build profile: -w ghc-9.4.8 -O1
In order, the following will be built (use -v for more details):
 - hello-flake-haskell-1.0.0 (exe:hello-flake-haskell) (first run)
Configuring executable 'hello-flake-haskell' for hello-flake-haskell-1.0.0..
Warning: Packages using 'cabal-version: >= 1.10' and before 'cabal-version:
3.4' must specify the 'default-language' field for each component (e.g.
Haskell98 or Haskell2010). If a component uses different languages in
different modules then list the other ones in the 'other-languages' field.
Warning: The 'license-file' field refers to the file 'LICENSE' which does not
exist.
Preprocessing executable 'hello-flake-haskell' for hello-flake-haskell-1.0.0..
Building executable 'hello-flake-haskell' for hello-flake-haskell-1.0.0..
[1 of 1] Compiling Main             ( Main.hs, /home/amy/codeberg/nix-book/source/new-
flake/haskell-flake/hello-haskell/dist-newstyle/build/x86_64-linux/ghc-9.4.8/hello-
flake-haskell-1.0.0/x/hello-flake-haskell/build/hello-flake-haskell/hello-flake-
haskell-tmp/Main.o )
[2 of 2] Linking /home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-
haskell/dist-newstyle/build/x86_64-linux/ghc-9.4.8/hello-flake-haskell-1.0.0/x/hello-
flake-haskell/build/hello-flake-haskell/hello-flake-haskell
$ cabal run
Hello from Haskell inside a Nix flake!
Your hostname is: wombat11k
$ exit
```

After a lot of output messages, the build succeeds and the program runs.

9.1.5. The Nix flake

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 would be different are the development shell and the package builder.

However, there's a simpler way, using `haskell-flake`.

flake.nix

```
{
  description = "a flake using Haskell";

  # This example uses haskell-flake to make things simpler.
  # See https://haskell.flake.page/ for more information and examples.

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixpkgs-unstable";
    flake-parts.url = "github:hercules-ci/flake-parts";
    haskell-flake.url = "github:srid/haskell-flake";
```

```
  };
  outputs = inputs@{ self, nixpkgs, flake-parts, ... }:
    flake-parts.lib.mkFlake { inherit inputs; } {
      systems = nixpkgs.lib.systems.flakeExposed;
      imports = [ inputs.haskell-flake.flakeModule ];

      perSystem = { self', pkgs, ... }: {
        haskellProjects.default = {};

        # haskell-flake doesn't set the default package, but you can do it here.
        packages.default = self'.packages.hello-flake-haskell;
      };
    };
}
```

The above definition will work for most of your haskell projects; simply change the `description` and the package name in `packages.default`. Let's try out the new flake.

```
$ nix run
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-haskell' is dirty
error: getting status of '/nix/store/0ccnxa25whszw7mgbgyzdm4nqc0zwnm8-source/flake.nix': No such file or directory
```

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 hello-flake-haskell.cabal Main.hs
$ nix run
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-haskell' is dirty
warning: creating lock file '/home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-haskell/flake.lock'
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-haskell' is dirty
these 2 derivations will be built:
  /nix/store/5ar7vhd4nz8wbqrsgaxqzkh6b4ggvsrv-source-hello-flake-haskell-sdist.tar.gz.drv
  /nix/store/hi70w0gzjfj213r0xhhva7n617hfa378-hello-flake-haskell-1.0.0.drv
building '/nix/store/5ar7vhd4nz8wbqrsgaxqzkh6b4ggvsrv-source-hello-flake-haskell-sdist.tar.gz.drv'...
error: builder for '/nix/store/5ar7vhd4nz8wbqrsgaxqzkh6b4ggvsrv-source-hello-flake-haskell-sdist.tar.gz.drv' failed with exit code 1;
       last 7 log lines:
       > unpacking source archive /nix/store/fdxvjgdpsfjrkanzbx43g1yxf2b1lp4b-source-hello-flake-haskell
       > source root is source-hello-flake-haskell
       > Config file path source is default config file.
```

```
       > Config file not found: /build/source-hello-flake-haskell/.config/cabal/config
       > Writing default configuration to
       > /build/source-hello-flake-haskell/.config/cabal/config
       > /build/source-hello-flake-haskell/./LICENSE: withBinaryFile: does not exist
(No such file or directory)
       For full logs, run 'nix log /nix/store/5ar7vhd4nz8wbqrsgaxqzkh6b4ggvsrv-source-
hello-flake-haskell-sdist.tar.gz.drv'.
error: 1 dependencies of derivation '/nix/store/hi70w0gzjfj213r0xhhva7n617hfa378-
hello-flake-haskell-1.0.0.drv' failed to build
```

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) 39ade13] initial commit
 4 files changed, 137 insertions(+)
 create mode 100644 Main.hs
 create mode 100644 flake.lock
 create mode 100644 flake.nix
 create mode 100644 hello-flake-haskell.cabal
```

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

```
$ cd ..
$ nix run ./hello-haskell
these 2 derivations will be built:
  /nix/store/5ar7vhd4nz8wbqrsgaxqzkh6b4ggvsrv-source-hello-flake-haskell-
sdist.tar.gz.drv
  /nix/store/hi70w0gzjfj213r0xhhva7n617hfa378-hello-flake-haskell-1.0.0.drv
building '/nix/store/5ar7vhd4nz8wbqrsgaxqzkh6b4ggvsrv-source-hello-flake-haskell-
sdist.tar.gz.drv'...
error: builder for '/nix/store/5ar7vhd4nz8wbqrsgaxqzkh6b4ggvsrv-source-hello-flake-
haskell-sdist.tar.gz.drv' failed with exit code 1;
       last 7 log lines:
       > unpacking source archive /nix/store/fdxvjgdpsfjrkanzbx43g1yxf2b1lp4b-source-
hello-flake-haskell
       > source root is source-hello-flake-haskell
       > Config file path source is default config file.
       > Config file not found: /build/source-hello-flake-haskell/.config/cabal/config
       > Writing default configuration to
       > /build/source-hello-flake-haskell/.config/cabal/config
       > /build/source-hello-flake-haskell/./LICENSE: withBinaryFile: does not exist
(No such file or directory)
       For full logs, run 'nix log /nix/store/5ar7vhd4nz8wbqrsgaxqzkh6b4ggvsrv-source-
hello-flake-haskell-sdist.tar.gz.drv'.
```

```
error: 1 dependencies of derivation '/nix/store/hi70w0gzjfj213r0xhhva7n617hfa378-
hello-flake-haskell-1.0.0.drv' failed to build
```

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 Haskell flake.

9.2. Python

Start with an empty directory and create a git repository.

```
$ mkdir hello-python
$ cd hello-python
$ git init
Initialized empty Git repository in /home/amy/codeberg/nix-book/source/new-
flake/python-flake/hello-python/.git/
```

9.2.1. A simple Python program

Next, we'll create a simple Python program.

hello.py

```python
#!/usr/bin/env python

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

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

Before we package 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.

> 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!
```

9.2.2. A Python builder

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.

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/qp5zys77biz7imbk6yy85q5pdv7qk84j-python3-3.11.6/bin/python: No module named
build
```

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.

9.2.3. The Nix flake

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.

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/codeberg/nix-book/source/new-flake/python-flake/hello-
python' is dirty
error: getting status of '/nix/store/0ccnxa25whszw7mgbgyzdm4nqc0zwnm8-
source/flake.nix': No such file or directory
```

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/codeberg/nix-book/source/new-flake/python-flake/hello-
python' is dirty
warning: creating lock file '/home/amy/codeberg/nix-book/source/new-flake/python-
flake/hello-python/flake.lock'
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/python-flake/hello-
python' is dirty
this derivation will be built:
  /nix/store/dh97mik5m481j14am9dgy8jx9ppv5b6d-hello-flake-python.drv
building '/nix/store/dh97mik5m481j14am9dgy8jx9ppv5b6d-hello-flake-python.drv'...
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) 6bec7c3] 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 ./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.

Chapter 10. Recipes

This chapter provides examples of how to use Nix in a variety of scanarios. Multiple types of recipes are provided are provided for some scenarios; comparing the different recipes will help you better understand Nix.

- An *"ad hoc" shell* is useful when you want to quickly create an environment for a one-off task.
- A *traditional nix shell* is useful when you want to define an environment that you will use more than once.
- *Nix flakes* are the recommended approach for development projects.
- You can use `nix-shell` to run scripts in arbitrary languages, providing the necessary dependencies. This is particularly convenient for standalone scripts because you don't need to create a repo and write a separate `flake.nix`. The script should start with two *"shebang"* (`#!`) commands. The first should invoke `nix-shell`. The second should declares the script interpreter and any dependencies.

10.1. Access to a top-level package from the Nixpkgs/NixOS repo

Ex: Access two packages from nixpkgs: hello and cowsay.

10.1.1. From the command line

```
$ nix-shell -p "[hello cowsay]"
$ hello
Hello, world!
$ cowsay "moo"
 _____
< moo >
 -----
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

10.1.2. In `shell.nix`

shell.nix

```
with (import <nixpkgs> {});
mkShell {
  buildInputs = [
    hello
    cowsay
```

```
  ];
}
```

Here's a demonstration using the shell.

```
$ nix-shell
$ hello
Hello, world!
$ cowsay "moo"
 _____
< moo >
 -----
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

10.1.3. In a Bash script

Script

```
#! /usr/bin/env nix-shell
#! nix-shell -i bash -p "[hello cowsay]"
hello
cowsay "Pretty cool, huh?"
```

Output

```
Hello, world!
 ___________________
< Pretty cool, huh? >
 -------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

10.2. Access to a package defined in a remote git repo

Ex: Access a package called `hello-nix`, which is defined in a remote git repo on codeberg. To use a package from GitHub, GitLab, or any other public platform, modify the URL.

10.2.1. In `shell.nix`

shell.nix

```
with (import <nixpkgs> {});
let
  hello-nix = import (builtins.fetchGit {
                                           url = "https://codeberg.org/mhwombat/hello-nix";
                                           rev = "aa2c87f8b89578b069b09fdb2be30a0c9d8a77d8";
                                         });
in
mkShell {
  buildInputs = [ hello-nix ];
}
```

Here's a demonstration using the shell.

```
$ nix-shell
$ hello-nix
Hello from your nix package!
```

10.3. Access to a flake defined in a remote git repo

Ex: Access a flake called `hello-flake`, which is defined in a remote git repo on codeberg. To use a package from GitHub, GitLab, or any other public platform, modify the URL.

10.3.1. In `shell.nix`

shell.nix

```
with (import <nixpkgs> {});
let
   hello-flake = ( builtins.getFlake
                     git+https://codeberg.org/mhwombat/hello-flake?ref=main&rev=3aa43dbe7be878dde7b2bdcbe992fe1705da3150
         ).packages.${builtins.currentSystem}.default;
in
mkShell {
  buildInputs = [
    hello-flake
  ];
}
```

Here's a demonstration using the shell.

```
$ nix-shell
$ hello-flake
```

```
Hello from your flake!
```

10.4. Access to a Haskell library package in the nixpkgs repo (without a `.cabal` file)

Occasionally you might want to run a short Haskell program that depends on a Haskell library, but you don't want to bother writing a cabal file.

Example: Access the `containers` package from the `haskellPackages` set in the nixpkgs repo.

10.4.1. In `shell.nix`

shell.nix

```nix
with (import <nixpkgs> {});
let
  customGhc = haskellPackages.ghcWithPackages (pkgs: with pkgs; [ containers ]);
in
mkShell {
  buildInputs = [
                  customGhc
                ];
}
```

Here's a short Haskell program that uses it.

Main.hs

```haskell
import Data.Map

m :: Map String Int
m = fromList [("cats", 3), ("dogs", 2)]

main :: IO ()
main = do
  let cats = findWithDefault 0 "cats" m
  let dogs = findWithDefault 0 "dogs" m
  let zebras = findWithDefault 0 "zebras" m
  print $ "I have " ++ show cats ++ " cats, " ++ show dogs ++ " dogs, and " ++ show zebras ++ " zebras."
```

Here's a demonstration using the program.

```
$ nix-shell
$ runghc Main.hs
"I have 3 cats, 2 dogs, and 0 zebras."
```

10.4.2. In a Haskell script

Script

```
#! /usr/bin/env nix-shell
#! nix-shell -p "haskellPackages.ghcWithPackages (p: [p.containers])"
#! nix-shell -i runghc

import Data.Map

m :: Map String Int
m = fromList [("cats", 3), ("dogs", 2)]

main :: IO ()
main = do
  let cats = findWithDefault 0 "cats" m
  let dogs = findWithDefault 0 "dogs" m
  let zebras = findWithDefault 0 "zebras" m
  print $ "I have " ++ show cats ++ " cats, " ++ show dogs ++ " dogs, and " ++ show zebras ++ " zebras."
```

Output

```
"I have 3 cats, 2 dogs, and 0 zebras."
```

10.5. Access to a Haskell package on your local computer

Ex: Access three Haskell packages (`pandoc-linear-table`, `pandoc-logic-proof`, and `pandoc-columns`) that are on my hard drive.

10.5.1. In `shell.nix`

shell.nix

```
with (import <nixpkgs> {});
let
  pandoc-linear-table = haskellPackages.callCabal2nix "pandoc-linear-table" /home/amy/github/pandoc-linear-table {};
  pandoc-logic-proof = haskellPackages.callCabal2nix "pandoc-logic-proof" /home/amy/github/pandoc-logic-proof {};
  pandoc-columns = haskellPackages.callCabal2nix "pandoc-columns" /home/amy/github/pandoc-columns {};
in
mkShell {
  buildInputs = [
                  pandoc
                  pandoc-linear-table
```

```
                  pandoc-logic-proof
                  pandoc-columns
                ];
}
```

10.6. Access to a Haskell package on your local computer, with interdependencies

Ex: Access four Haskell packages (`pandoc-linear-table`, `pandoc-logic-proof`, `pandoc-columns` and `pandoc-maths-web`) that are on my hard drive. The fourth package depends on the first three to build.

10.6.1. In `shell.nix`

shell.nix

```
with (import <nixpkgs> {});
let
  pandoc-linear-table = haskellPackages.callCabal2nix "pandoc-linear-table" /home/amy/github/pandoc-linear-table {};
  pandoc-logic-proof = haskellPackages.callCabal2nix "pandoc-logic-proof" /home/amy/github/pandoc-logic-proof {};
  pandoc-columns = haskellPackages.callCabal2nix "pandoc-columns" /home/amy/github/pandoc-columns {};
  pandoc-maths-web = haskellPackages.callCabal2nix "pandoc-maths-web" /home/amy/github/pandoc-maths-web
                       {
                         inherit pandoc-linear-table pandoc-logic-proof pandoc-columns;
                       };
in
mkShell {
  buildInputs = [
                  pandoc
                  pandoc-linear-table
                  pandoc-logic-proof
                  pandoc-columns
                  pandoc-maths-web
                ];
}
```

10.7. Access to a Python library package in the nixpkgs repo (without using a Python builder)

Occasionally you might want to run a short Python program that depends on a Python library, but you don't want to bother configuring a builder.

Example: Access the `html_sanitizer` package from the `python3nnPackages` set in the nixpkgs repo.

10.7.1. In a Python script

Script

```python
#! /usr/bin/env nix-shell
#! nix-shell -i python3 -p python3Packages.html-sanitizer

from html_sanitizer import Sanitizer
sanitizer = Sanitizer()  # default configuration

original='<span style="font-weight:bold">some text</span>'
print('original: ', original)

sanitized=sanitizer.sanitize(original)
print('sanitized: ', sanitized)
```

Output

```
original:  <span style="font-weight:bold">some text</span>
sanitized:  <strong>some text</strong>
```

10.8. Set an environment variable

Ex: Set the value of the environment variable FOO to “bar”

10.8.1. In `shell.nix`

shell.nix

```nix
with (import <nixpkgs> {});
mkShell {
  shellHook = ''
    export FOO="bar"
  '';
}
```