A generic flake template

Note: This is a "template" in the sense that it is a useful pattern to follow. It's not something you can use with `nix flake init`. I may create a set of templates like that in the future; watch this repo.

```
{
  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;
        };
      }
    );
}
```

---

❶ See this guide for help on adding flake dependencies.

❷ Any dependencies in the input section should also be listed here.

❸ 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 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.

❹ Here you would list any tools (e.g., compilers and language-specific build tools) you want to have available when doing development on the package. 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. I recommend that you do a web search for "nix *language*", and try to find resources that were written or updated recently. If you don't need a special development environment, you can omit this section.

❺ The package definition 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.

Here are a few functions that are commonly used:

General-purpose: mkDerivation Handles the standard ./configure; make; make install scenario, customisable.

Python: buildPythonApplication, buildPythonPackage.

Haskell: mkDerivation (Haskell version, which is a wrapper around the standard environment version),developPackage, callCabal2Nix.