Creating isolated development environments with Virtualenv

The traditional approach to developing applications in Python is to isolate these applications in a virtual environment. This is done to avoid installing packages globally and reduce conflicts during application development.

A virtual environment is an isolated environment where application dependencies installed can only be accessed within it. As a result, the application can only access packages and interact only within this environment.

Creating a virtual environment

By default, the `venv` module from the standard library is installed in Python3. The `venv` module is responsible for creating a virtual environment. Let's create a `todos` folder and create a virtual environment in it by running the following commands:

```
$ mkdir todos && cd todos
$ python3 -m venv venv
```

The `venv` module takes an argument, which is the name of the folder where the virtual environment should be installed into. In our newly created virtual environment, a copy of the Python interpreter is installed in the `lib` folder, and the files enabling interactions within the virtual environment are stored in the `bin` folder.

Activating and deactivating the virtual environment

To activate a virtual environment, we run the following command:

```
$ source venv/bin/activate
```

The preceding command instructs your shell to use the virtual environment's interpreter and packages by default. Upon activating the virtual environment, a prefix of the `venv` virtual environment folder is added before the prompt as follows:

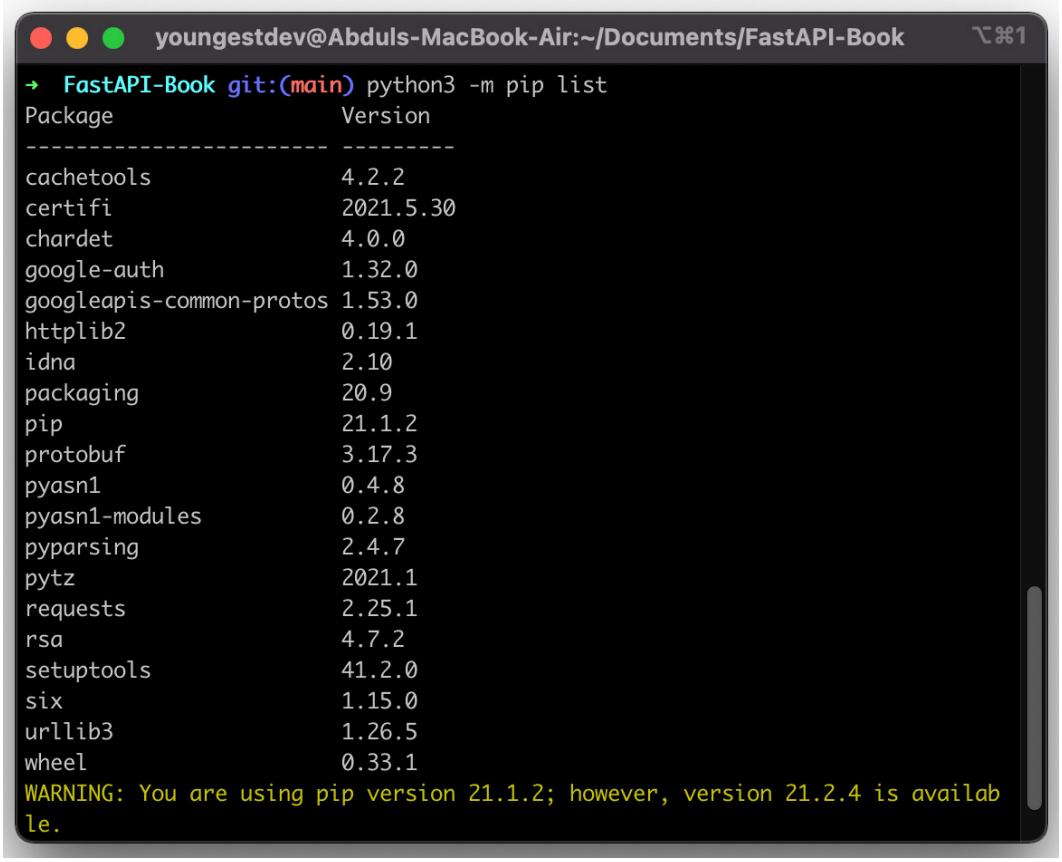

Figure 1.3 – Prefixed prompt

To deactivate a virtual environment, the `deactivate` command is run in the prompt. Running the command immediately exits the isolated environment and the prefix is removed as follows:

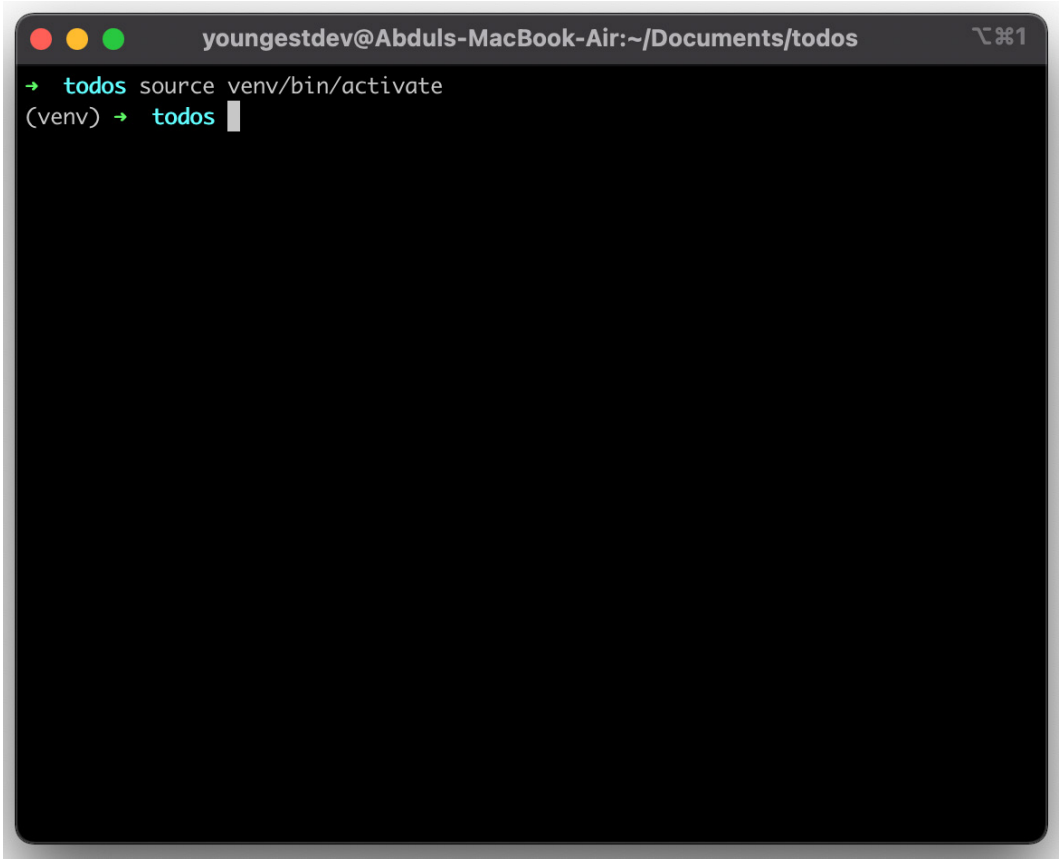

Figure 1.4 – Deactivating a virtual environment

> **Important Note**
> You can also create a virtual environment and manage application dependencies using *Pipenv* and *Poetry*.

Now that we have created the virtual environment, we can now proceed to understand how package management with **pip** works.

Package management with pip

A FastAPI application constitutes packages, therefore you will be introduced to package management practices, such as installing packages, removing packages, and updating packages for your application.

Installing packages from the source can turn out to be a cumbersome task as, most of the time, it involves downloading and unzipping `.tar.gz` files before manual installation. In a scenario where a hundred packages are to be installed, this method becomes inefficient. Then, how do you automate this process?

Pip is a Python package manager like JavaScript's `yarn`; it enables you to automate the process of installing Python packages – both globally and locally.

Installing pip

Pip is automatically installed during a Python installation. You can verify whether pip is installed by running the following command in your terminal:

```
$ python3 -m pip list
```

The preceding command should return a list of packages installed. The output should be similar to the following figure:

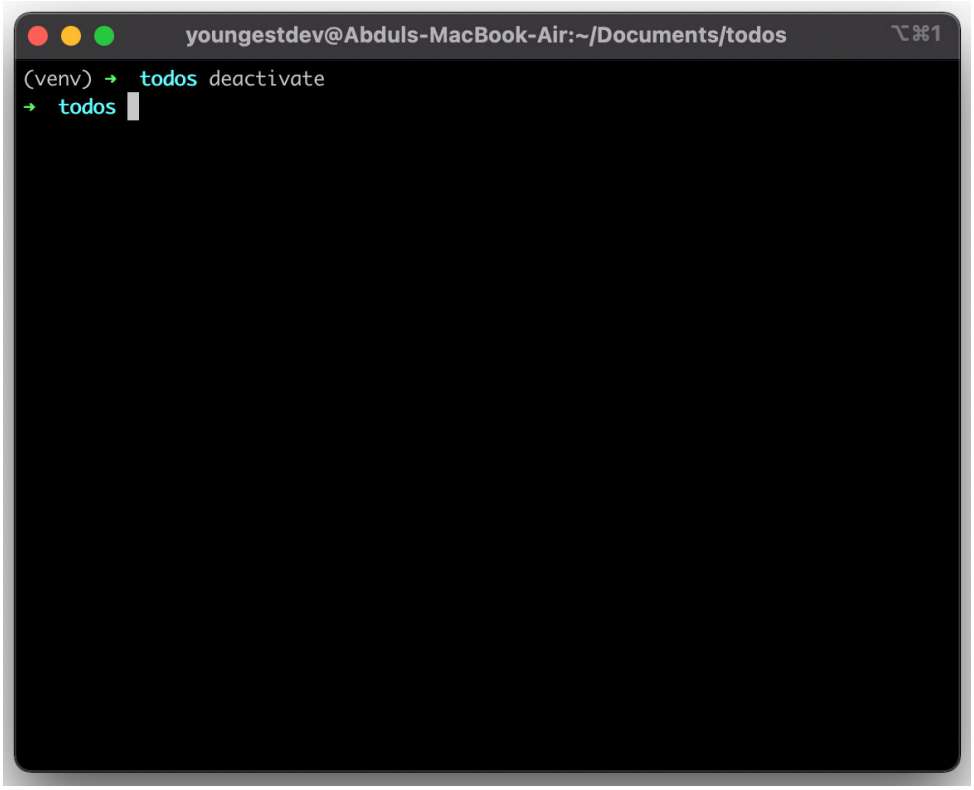

Figure 1.5 – List of installed Python packages

If the command returns an error, follow the instructions at `https://pip.pypa.io/en/stable/installation/` to install pip.

Basic commands

With `pip` installed, let's learn its basic commands. To install the `FastAPI` package with pip, we run the following command:

```
$ pip install fastapi
```

On a Unix operating system, such as Mac or Linux, in some cases, the `sudo` keyword is prepended to install global packages.

To uninstall a package, the following command is used:

```
$ pip uninstall fastapi
```

To collate the current packages installed in a project into a file, we use the following `freeze` command:

```
$ pip freeze > requirements.txt
```

The `>` operator tells bash to save the output from the command into the `requirements.txt` file. This means that running `pip freeze` returns an output of all the currently installed packages.

To install packages from a file such as the `requirements.txt` file, the following command is used:

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

The preceding command is mostly used in deployment.

Now that you have learned the basics of pip and have gone over some basic commands, let's learn the basics of **Docker**.

Setting up Docker

As our application grows into having multiple layers, such as a database, coupling the application into a single piece enables us to deploy our application. We'll be using **Docker** to containerize our application layers into a single image, which can then be easily deployed locally or in the cloud.