Managing Python Packages and Virtual Environments

by Reindert-Jan Ekker

Course Overview

Course Overview

Hi everyone! My name is Reindert-Jan Ekker, and welcome to my course, Managing Python Packages and Virtual Environments. I'm a senior developer and educator. And in this course, I'll teach you about managing Python packages and project requirements. This is a very important skillset for any Python coder who goes beyond simple scripts and starts writing larger projects. We'll learn about Pip, the Python package installer, virtual environments, which allow us to create Python projects with their own dependencies, virtualenvwrapper, a popular tool that makes working with the other tools a lot easier, and we'll take a look at current developments in the Python world and some new tools that you should know about. By the end of this course, you will have all the necessary skills to manage the dependencies for your Python projects and share those dependencies with your teammates. Before beginning the course, you should be familiar with the basics of Python programming. I hope you'll join me on this journey to learn Python project and package management with the course, Managing Python Packages and Virtual Environments, at Pluralsight.

Managing Python Packages with pip

Introduction

Hi. My name is Reindert-Jan Ekker. Welcome to this course that will teach you how to manage Python packages and virtual environments. In this module, we'll start with the one tool that we cannot live without, and it's called pip. Anyone who writes more than just a few lines of Python code will at some point need to use a third-party library. As an example, you might want to create your company's home page using Flask, a Python framework for creating websites. In this case, Flask could be one of the dependencies for your projects. This basically means that your code calls functions from Flask or uses its classes or uses it in some other way. The important point here is that your project now depends on Flask to be present so that you can create a working product. This means you want to somehow get the source code for Flask so that you can use it. For this, Python provides ways to create packages, which make it possible to bundle and transport a set of Python code and upload it somewhere to the internet where a user can find, download, and install it so you can then write code that uses that project. At some point during the development process, you might want to upgrade Flask to a newer version or maybe remove it if you decide to use something else instead. In other words, we want to be able to manage this set of installed packages on our system. Now most real world developers have more than a single Python project that they work on, and each of these might have its own dependencies, and to handle this, there is something called virtual environments, which give us a way to install project dependencies separately for each project. Of course, we want to see how to create, use, and manage these environments, and what the best practices and recommended tools are because the world of Python dependency management is frankly quite messy right now. And I want to give you an overview of what the important and popular tools are and where the Python community is headed. By the way, this course is strictly about managing the dependencies of your Python project and not about creating a package for your finished product and shipping that. That's a more advanced subject for a different time. So how is this course organized? Well, I'll start with the most basic and most important tool you need, and that's the pip package manager. There's actually quite a lot to say about it, and we'll have a whole module dedicated to it, and once you know how to use pip, we'll talk about virtual environments, which enable you to manage dependencies specifically for a single project. In the module after that, we will look at virtualenvwrapper, a great tool for managing your virtual environments in a more convenient way.

In the final module, I will tell you about other and newer tools that you might want to work with or that you should at least know about. This course is for those of you who are relatively new to the Python programming language and are moving from simple scripts to larger projects, which will mean that your projects will contain code probably organized into multiple files, and you'll be starting to solve more complex problems. To do so, you'll probably start using third-party modules and libraries. In other words, you'll use code written by someone else to solve part of your problems. The only things that you need to have already to be able to understand everything in this course is basic knowledge of the Python programming language and Python installation on your system. If you have both of those, you're good to go. Now before we continue, let me clear one thing up. The word package in Python can mean two things, a directory containing a dunder __init__py file, and this is a way to structure your Python code, and you might have one or more of these packages as a part of your Python projects. You use code from packages by importing them into your Python code. Now for the purpose of this course, when I say the word package, I will not be talking about this kind of package. Instead, I will be talking about a different kind of package, a Python distribution package, which is, to quote the Python package authority, "a versioned archive file that contains Python packages, modules, and other resource files that are used to distribute a release. " The archive file is what an end-user will download from the internet and install. So with a package, I denote some way to send a bunch of Python code to someone else across the internet. Now this course focuses on working with these packages as a user, which means that things like creating these packages, what kind of file format they use, and how to upload them online are beyond the scope of this course.

Getting Started

So with that said, let's get started with some hands-on demos. I'll start by showing you a simple project with a third-party dependency, which doesn't run because we still need to install that dependency. To do so, we need pip, and we'll see how to check whether it's present on our system. Let's say I'm starting a new project where I will need to retrieve some currency data from the internet. One of the more popular libraries that can do that for us is called Requests. And we can see its home page right here. And here's a little script that will get the rate of the dollar against the euro for me. To use the Requests library, I start by importing requests. To retrieve data over HTTP, I then call requests.get and pass a URL. In this case, I'll use the API published by the European Central Bank, and I'm asking for the latest euro to dollar exchange rate. The answer from the server is stored in the variable response, and I get the text from there and print it in the last line. So far, so good. Now let's try to run this script from the command line. So to run this, I

run python get_rates.py, which is the name of my script, and this gives me an ImportError: No module named requests. Of course this is because we haven't actually installed requests on our system yet. Now if you are on Windows, and you get a command, Not found, when you run this, you probably didn't put Python on your path, and we'll see how to fix this in a moment. To install a package like Requests on our system, we would normally run pip, the Python package manager, and tell it to install Requests. But let's first see if we can run pip at all. Please type along with me and run the following command on your system, pip --V, and make sure to use a capital letter V here. Now if you've correctly installed pip, you should see something similar to this. Pip will show you the installed version and location. You might get a completely different version. For example, on another Linux machine, I have pip version 18 installed. But that doesn't matter right now. If you get any output like this from pip, you're using some version of pip, and that's good enough for now. In that case, you might want to choose to skip the next few clips about installing pip because you don't need to and jump to the part where I start using pip to install a package. If, on the other hand, you get an error like Command not found, we'll have to make sure you have pip installed. And let's take a moment to look into that. Here's a general rule. Pip is the standard tool for installing Python packages and, as such, it comes preinstalled with all recent versions of Python. To be precise, those are versions 2.7 .9 or later when you use Python 2 or version 3.4 or later when you use Python 3. Unfortunately, though, you can only really count on pip being preinstalled when you downloaded your Python from python.org and installed it yourself. In case you use an older version of Python, this will not have pip installed by default, and you will need to install it yourself. You can do so by browsing to pip-- installer.org and following the installation instructions there. Now, again, this will certainly work if you downloaded your Python version from python.org, but if, for example, you installed this older Python version using a Linux package manager, there's a better way, and we'll cover that in a moment in the Linux-specific part of this module. Now in the next three clips, I will cover how to get a working pip on Windows, then on Mac, and, finally, on Linux. So if you want, you cannot skip ahead to the part of the module that fits your system, or skip the installation part entirely if you already have a working pip.

Demo: Installing pip on Windows

Next, I'm going to show you a short demo where we'll see how to configure your path variable so that pip runs correctly. And, of course, we'll check that it works by actually running pip. So, assuming you're using a relatively new Python version and you try to run pip but it didn't work, how do we fix this? Well, if you're using Windows, you probably downloaded Python from no python.org, and there's probably no need to install pip in a separate step. But there's still a

chance that you got a Command not found error when you tried to run pip a moment ago. If that happened to you, then you probably didn't add pip and Python to the path correctly. So let's see how to fix this. So one important thing when installing Python is that in the installer, you should enable this checkbox here, which will add both Python and pip to your path. And that lets you run them from the command line. Now if you forgot this when installing Python, then if you try to run pip, you will get an error message. Now let me show you how to fix this. First, you need to find out where your Python is installed. Usually, this will be either at the root of your file system, so most of the time this will be the root of the C drive, and there should be a folder here called Python 2.7 or something similar. As you can see, I don't have such a folder, but there's another likely location, and that is in my Users home folder. And in here, I go into AppData, Local, Programs, and here you find a folder Python, which contains any Python versions you installed like, in my case, there's a Python 3.7 here. So once you've located your Python installation, you move into that folder, and then click the address bar here and copy the location. Now we want to add this location to our path variable. And to do so, I first want to locate the This PC icon, which is here on the left. And I do a right mouse button click and then click Properties. Next, I choose Advanced system settings, and then this new window pops up where I can click the button, Environment Variables. So here you see the variables defined for your PC. And here's the variable called path. If it doesn't show here, you can add it. In my case, it's there already, and I'll click Edit. Now in this window, I can add locations to the path variable, and I'll add the location of the Python installed that I copied a moment ago. So this will put the Python executable itself on the path. But to be able to use pip, we need to add another location. So I'll add another line and paste the same location in here. But this time I'm adding the name of the folder that actually contains pip. So I'm typing a backslash and then Scripts with a capital S. So with that done, we click OK and OK and OK, and then we have to restart the command line application. So let me close this window and start a new command line. And now when I type pip -V, we see that it works. By the way, if this doesn't work for you, try restarting your PC because in some cases, Windows seems to refuse to update the path variable. So now we know how to make pip work on Windows. Let's move on to other operating systems.

Installing pip on Mac OS or Linux

On Mac OS, there's a Python version installed by default. But this is meant for running system scripts and services and not for development. I would recommend installing your own Python either by using the Homebrew package manager, which will install pip automatically, or by downloading from python.org. Since both of these are pretty straightforward and should come

with a working pip, I will not show this in a demo because I trust you will be able to do this by yourself. On Linux, however, the situation is quite complicated. And I'm afraid I cannot tell you how to install pip for your specific machine. But in general, there will already be some Python version installed, most probably Python 2, and this may or may not include pip. So you might want to install Python 3 and pip or only pip. For both of those, I recommend that you use your system's package manager. You will need to search for the correct names of both Python and pip for your distribution, but that shouldn't give you any trouble. Now if you use your package manager, this will install a pip version that has been tested for your distribution and has things like security patches. And it might not be the latest version of pip. But this is the correct version for your system, so you shouldn't upgrade it. We'll soon see how to upgrade the version of pip you are using in your projects instead without messing with the system pip that your system package manager has installed.

Demo: Linux Installation

So let's have a short look at how this looks in practice on a Linux machine. On a Mac, it will be very similar when you use Homebrew. We'll install pip and run it, and then we'll install pip for Python 3 and run that. So here I am on the Linux command line, and in this case, I have a machine that doesn't have pip install. Now the official pip home page will tell you to run a Python program to install pip, but the correct way is actually to use your package manager. This is a Debian system, and, in this case, I install it by saying sudo, which means I want to run this as a super user, and then apt, which is the packet manager, and I'm going to tell it to install a package called python-pip. And after that, we see that now we have pip 9 for Python 2.7. This is actually a pretty old pip version, but it's what's been tested to work correctly for this Debian system. So let's not mess with it. The system default Python version for Debian is 2.7. Now what if we want to use Python 3 instead? Well, Debian does come with Python 3 installed by default, so I don't have to do anything here, but on another system, you might need to do something like yum install python3 if you're on Red Hat for example. I can see that Python 3 is on my system by running python3 -V, which tells me that I have Python 3.5 installed. Now the currently installed pip tells me that it will install packages for Python 2.7. So for Python 3, I will need another version of pip, and I can get that by saying apt install new python3-pip. And, of course, I shouldn't forget to say sudo because I need to run this as an administrator. So this installs another version of pip, and it's installed as the command pip3. Let me show you. So the point of this is that if you have multiple Python versions, which is very common on Linux, you will need a pip for each version. On other systems, this is not a thing because there you would usually install Python with pip included. But

most Linux distributions package Python and pip separately, so you will need separate install steps.

Demo: Package Management with pip

So now we've come to the point where I'm going to assume that you have a useful pip on your system. So let's go ahead and see how to actually use pip. And the most important use case is, of course, installing Python packages, as well as the opposite operation, removing packages. We'll also learn how to list the packages that have been installed, inspect packages, how to get help if you need to know more about pip, and how to search for Python packages. Before we start actually using pip, let me take a moment to point out an important rule. When you install packages, you always want to work inside a virtual environment. In the next module, I'll show you how to do exactly that, and we'll put the things we've learned so far in practice. Right now, I just think it's important that you don't start installing lots of packages and mess up your Python environment. Following this rule will help you keep things nice and clean. Now on some systems, if you try to run pip, you won't be allowed to if you don't have administrator access. On Mac and Linux systems, you might try to use the sudo commands to run pip. And that would probably work. But it's a bad idea because you will be installing things system-wide, and you might mess up your system. The consequence of these rules is that you might not want to work along with the next demo just yet. Just sit back for a moment. We'll start doing some real work pretty soon. What you see here is the command line on my Debian Linux system. Now remember that trying to run my program didn't work because Requests wasn't installed. Well, just to be sure, let's type pip list, which shows me the list of all currently installed packages with their versions. And on this system, there are several packages already installed by default. On your system, this might show a completely different list of packages, and that's okay. So to make my program run, I want to install the package Requests. And before you type along with me, let me just say that you might not want to do this at all. So please just wait for a moment before you do anything. This is the command we need to run, pip install requests, and when I press Enter, you'll see that pip downloads and installs the latest version of this package for us. But that's not all. It also installs any other packages that Requests depends on like certifi, idna, and urllib3. So after this, when we run pip list, we see that now the latest versions of these packages have been installed. So why did I warn you not to run pip install like I did? Well, in a sense, I've just polluted my own Python environment. Every Python project that you now create will run in this environment and have access to these packages we just installed. And this gives rise to a number of problems, which I'll talk more about in the next module. For now, you should just know this, please never just use pip

install like I did. Always do it inside a virtual environment. Now what that means and what a virtual environment is I will explain in the next module. For now, let's focus on pip. Just for the sake of this demo, let's say that I want to remove the packages I just installed. I can uninstall something by saying pip uninstall requests. And this removes the Requests package. But the thing to realize is that unlike the install command, this action doesn't include the dependencies of Requests. So if I do a pip list now, we see that the other packages that were installed with Requests are still there. So I would have to uninstall them one by one, or I can list multiple ones on the same line. So after I do this, I'm back in the same situation that I had before I started the demo.

Demo: Getting Package Information with pip

Now pip can do more than the few things we just saw, and I will not show everything in this module. So what if you want to know more? Well, of course, there's a nice user guide here on the pip home page, but pip offers some help by itself as well. You can look at it by saying pip help, and it shows you all of the commands and options it supports. If you want to know more about a specific command, you can ask help for that as well, for example, by saying pip help list, and this will show the documentation for the list command, and we can see some extra functionality that I didn't talk about yet. For example, you can list outdated packages with -o. Let's try this. This shows us all outdated packages, and it shows us that Debian by default installs older versions of certain packages. Now the fact that your system might depend on older versions of packages is one reason we don't like to install system-wide stuff with pip because we don't want to interfere with this system setup. Instead, we'll learn how to use virtual environments in the next module. Now in case I want to know a little more about these packages here, I can use the command pip show. Let's say pip show six, and let's see what the package six is for. And apparently it holds compatibility code for Python 2 and 3. And if we want to know more, we can visit the URL shown there. Now when I install a Python package, where does pip actually download it from? Well, there's this site called PyPI or the Python Package Index. And this is a repository for Python packages. You can use it to search for packages to use in your projects, or, when you create a package, you can host it on PyPI. By the way, this site is also called the cheese shop after a famous sketch by Monty Python and because the packages you download are in a format called a wheel, like a wheel of cheese. So let's say I'm searching for packages that help me implement a neural network. I might start here and search for neural network. And here on the left, let me specify that I want something that is ready for production. So here's a list of packages that match that. Now if I would select one of these packages, let's say Keras, then if I would ask pip to install

that package by saying pip install Keras, it would download it from the cheese shop and install the latest version of that package.

Review

We've seen how to use pip to install packages, and you can do this by saying pip install followed by the name of the packages you want to install. This can be either a single package or multiple packages separated by spaces. In a very similar way, you can remove packages with the uninstall command, again either naming a single package or multiple ones. We've also seen how to find out information using pip. We can list all installed packages in the system using pip list, and this will show you all installed packages with their versions. You can also get some more info on a specific package by saying pip show. If you want more detailed information about the features that pip has to offer, you can say pip help. Or if you want to have help on a specific command, you can get help for that command by saying pip help followed by the command. So here pip help install will give me some extra info on all the options that pip install supports. Another way to find help about pip is, of course, to visit its home page, pip- installer.org. Finally, there's this site called the cheese shop, pypi.org, which is short for the Python Package Index, and this is where all the packages that you can download are hosted and where you can search for them if you want to find a package.

Demo: Where Are Packages Installed?

Now let's take a moment to take a closer look at what actually happens when we install a package. Where do the actual files end up? In this demo, we'll learn about the sys.path variable, and I'll show you what happens when you install a package for different versions of Python. So we have a script here that imports a module called Requests, and we've seen how to install that package. And, by the way, we can now also run this script. Now all of that's very nice, but what exactly happens when I run the import statement above? Where's my package installed, and how was it found? Well, there's a Python variable I can inspect to find out. Let's start a Python interpreter and type import sys. And then we can ask for the contents of sys.path. As you can see, this is a list of directories. These are the directories the Python interpreter searches to find the module you're trying to import. On your system, they will probably be different, but the principal will be the same. The first item in the list is an empty string, which denotes the local directory from which the Python interpreter is being run. If what we're looking for is not in there, it continues with the other directories in the list. Now we can try searching through all of these

locations, but something else we can do is to inspect the package. And this tells me that the request has been installed in the .local folder in my home directory. Again, the actual location might differ for you depending on lots of things like your OS, Python version, and more. So at least now I know where pip puts the packages I installed. But if we look at all the paths we've seen in the demo so far, you might have noticed that all of them were specific for this Python version, 2.7, and that means that if I tried to run my program with Python 3, we get the input error we've seen before, and that's because if I want to run this program using another Python version, I have to install its dependencies separately for that Python version. I can do that with the pip that was installed with Python 3 called pip3. So let's do pip3 install requests. And now we can run the program with Python 3. Now let's check where the package was installed in this case by calling pip3 show requests. And in this case, the package also ends up in .local but in a directory reserved for installed packages for this Python version.

Demo: A Better Way to Call pip

In the final demo of this module, I want to show you another way to call pip that makes several things easier. As we've just seen, the pip command doesn't always install packages for the Python version you want. I know by now that installing something by saying pip install, let's, for example, install Flask, this will install something for Python 2, but that's only the case because that happens to be how this system is configured. On another system or even for another user on this system, there's no sure way to know for what Python version this will work unless we know the configuration of the system. That's why the preferred way to call pip is actually like this, python2 -m pip install flask. This does exactly the same, but in this case, I'm explicitly specifying the Python version I want to install with. And this will run the correct pip module that was installed for that Python version. I can even call python with the minor version number, 2.7 in this case. Calling pip like this can even solve some problems. Let me show you one case that probably only works on my system. I will do something I told you explicitly not to do, and that is to run pip outside of a virtual environment to upgrade itself. This will cause a problem because of the way this particular Debian system is configured. Pip was installed by the system package manager, but calling pip like this will install a new pip package for my user. But the pip command still points to the old system-wide pip. This causes a mismatch, and nothing seems to be broken. But, interestingly, if I call pip in the better preferred way, it works. The reason is that if I do it this way, Python will find the newly installed Python version for my user, whereas the pip command still points to the old system-wide version. So all of this demonstrates two things, this way of calling pip is actually better, and don't use pip to interfere with the packages that were installed by your systems

package manager. That brings us to the end of this module. We've started by testing for the presence of pip and how to install it if it's missing. Then we learned how to use pip to manage our packages. The commands I've shown you are pip install and uninstall for installing and removing packages, pip list and pip show for finding info about packages, and we've seen a short demo of the cheese shop where the packages are hosted. I've also talked about the location where pip puts the packages on your system. And we've seen the preferred way to call pip by saying python -m pip, which is the way I will use pip in the rest of this course. Very well! Let's move on to the next module and start creating some real projects using virtual environments, and we'll put all of this in practice.

Setting up Your Project with Virtual Environments

About Virtual Environments

Hi, and welcome to this module where I'll tell you how to set up a Python project and manage its dependencies using virtual environments. I'll start with giving you a little background, explaining why virtualenv is necessary and what it does. Of course, we'll see how to create virtual environments and we'll take a look at what's inside. Then I'll show you how to use these environments when working on your projects and how to specify the packages that your project depends on. In the previous module, we saw how to install third-party Python packages with pip and how pip by default installs everything system-wide. But there are a lot of potential problems with this approach. Just to name a few, when you're working on multiple Python projects that depend on different versions of the same library, how do you manage that? Or suppose you're on Linux and system scripts depend on a specific version of a library. If you need a different version of that library for your project, you might break your system. Or maybe you're on a multi-user system where different users have Python projects with different dependencies. Or what about a project which you need to test against several different Python versions or against various versions of some other library. Fortunately, there's a solution for all of our problems, and it's called virtual environments. A virtual environment is an isolated context in which we can install Python packages so that they cannot interfere with the dependencies for other projects or any system-wide package installs. From now on, my advice to you is to always work inside a virtual environment. Never do any global package installs anymore. It's good practice to create a virtual

environment every time you start a new project. You can then install all the dependencies for that project inside the virtual environment so they won't cause conflicts with other projects.

Creating a Virtual Environment

Let's go ahead and see how this looks in practice. I'll take you through the steps of starting a new Python project, creating a virtual environment, and we'll see what's actually inside the environment. So let's talk about starting a new project. I already wrote a tiny bit of code in get_rates.py here, and as we know, this only works when we have Requests installed. Now the correct way to handle this is to start a virtual environment for this project. We need a tool to do this, and it's called virtualenv. This is one of the few times we actually want to install something globally, so let's do this. So here I'm calling pip to install virtualenv. Now I should be able to call the virtualenv command like this, virtualenv --version. But as you can see, on this Linux system, it doesn't work. On most systems, this won't be a problem, but if you do get this same problem, you can probably fix this by uninstalling and then reinstalling globally as an administrator for this one time. So here I'm calling python -m pip with the sudo command, so I'm running this as an administrator. Although I did warn you not to do this, in this specific case, I think it's an acceptable solution. So now we have the virtualenv too, I'm going to create a new virtual environment. And the virtual environment itself is not part of my project. So I'm going to put it in a separate location. I'm going to make a new directory where I'm going to put all of my virtual environments. First, I'm going to move into my home directory. And I'm going to create a directory called virtualenvs. And to create a virtual environment in here, I can say virtualenv rates where rates will be the name of my new environment. Now this will take a moment, and what it does is it creates a whole new directory called rates. Let me show you. So here I am in my home folder in my File Manager, and as you can see, here's the virtualenvs folder I just created. Moving in there, here we have my new virtual environment called rates, which I just created with the virtualenv command. Moving in there, there's something here that looks a lot like a complete Python installation. There's a lib folder, which can hold things called packages, and a bin folder, which if you are on Windows will be called scripts, and let's take a look in there. And in here we find a Python interpreter, as well as a pip installer. So this means that we now have a dedicated Python and a dedicated pip for our projects. And there's also something in here called activate, which is a script we have to run to activate the environment. And we'll see about that in a moment. Now before we go into that, let's set up an environment for Python 3. We can do so using the same command, but this time I add an option, -p, to tell virtualenv which Python interpreter I want to use. I pass python3 as the value. And running this, going back to the File

Manager, we now have a new directory called rates_py3, and in there, there's also a lib folder and a bin folder. And in the bin folder this time, again, there's an activate script and a pip and a Python interpreter. But this time, it's a Python 3 interpreter. Very well. So now we know how to make a virtual environment. Let's see how to use it.

Working Inside a Virtual Environment

Now that we have a virtual environment, we are ready to learn how to use it. We'll have to start by activating the environment. And then we can run pip and Python, install packages, and when we're done working, we can deactivate the environment. In this demo, I will be working with the python3 environment we just created named rates_py3. Before we can start working, we need to activate this environment. And on Windows, you do this by running rates_py3\Scripts\ activate.bat. The first word here is the name of the virtual environment, rates_py3, and we run the activate.bat script inside the Scripts folder in there. Now make sure to use backslashes and a capital S in the word Scripts. Obviously for me, this isn't going to work because I'm not on Windows. On Linux or Mac OS, I do the same thing with the dot command typing. (space)rates_py3/bin/activate. Here the dot means we want to import some shell script code from a file, and that file is in my environment there, rates_py3, under bin/activate. After running it, we can see that the environment is active because the prompt has changed. You can see the name of the active environment at the start here. And this will be the same on Windows. Inside the environment, the Python command refers to the Python interpreter from inside the environment. So saying python -V, we see that the command python now refers to Python 3.5, even though usually it refers to Python 2 on this system. Similarly, pip now refers to the pip inside the environment. Let's do a pip list, and because it's best practice, we call pip as a module saying python -m pip list, and we see a very short list. Now as you may remember from the previous module, I have quite a list of packages installed by the system. And none of those show up here because we are inside the environment and only see the packages inside the environment. This also means that we cannot run our program right now. Let me show you. First, I'm going to go back to the demos folder, and we cannot run our program because Requests is not installed. Let's fix that. After calling python -m pip install requests, now our program works. And looking at the installed package, we can see that it's installed inside the actual virtual environment. Now if we deactivate the environment, we see the prompt change back, and Python now again points to my system default Python. Of course, on your system, this might be the same Python version as inside the environment, but please understand that each environment has a reference to its own

interpreter. Doing a pip list now shows a different list of packages. And although I did install a version of Requests before, this is located in a different location.

Review

To create a new virtual environment, you run the virtualenv module by saying virtualenv followed by the name of the environment you want to create. This will create the folder for the environment containing its own Python interpreter, pip, and more. Of course, you shouldn't forget to install virtualenv before you do this. You should know that there's a similar package included with newer versions of Python called venv. You can use it by calling python -m venv followed by the name of your environment. It doesn't work with Python 2, though, and it isn't as well supported by other tools as virtualenv is, although I expect it to become more popular in the future. For most purposes, it doesn't really matter whether you use virtualenv or venv, though, so if you're using Python 3, this might save you an install step. By the way, you might see yet another tool called pyvenv mentioned in some older documentation, and it does pretty much the same thing, but it's now deprecated, so don't use it. Once you have venv, regardless of which tool you used to create it, you need to activate it before you can use it. On Linux and Mac OS, you use the dot command followed by the name of your virtual environment, which in this example is myvenv. And then you say /bin/activate. On Windows, it's a bit different. You call the activate script directly without using a dot before it, and it's in a directory called Scripts, not bin, with a capital S. And, of course, don't forget that on Windows, the slash is pointed the other way. After activation, you can see the name of the active environment in the shell prompt, and you can start working. Once you're done, deactivate the virtual environment with the deactivate command. By the way, if you want to remove a virtual environment from your system, you can simply delete its folder. There are no uninstall steps needed at all. So when you're inside an active virtual environment, the Python command will point to the Python that's installed inside the environment, so you don't have to worry about calling the right version of Python anymore. And the same is true for pip. It will be the correct pip, and it will install packages inside the environment so that they will not cause any problems with your other projects or with the system installation.

Managing Project Requirements

Suppose you work in a development team. How do you make sure that each of you has the same packages installed in their virtual environment? I'll show you how in this demo. I made my project slightly more complex by importing another library called Box. The currency data we get from a

server is in JSON format, and that can convert it to a Python data structure by calling .json on it. And then I put this in a Box wrapper, which allows me to retrieve the data inside it on the next line by saying b.rates .USD. It's almost kind of silly to use Box here, but I'm just using it so we have an extra dependency for our project. To run this program, now I first have to install Box. And, of course, before we install anything, we need to check that we are inside an active environment, which we aren't. So let me call the activate script. I can actually refer to it from here like this. So the little squiggly character here at the start of the path refers to my home directory, and this will work on Linux and on Mac OS but not on Windows. So let's activate this. And now we're in an active environment, I can install Box by saying python -m pip install python-box, which is the name of the Box package. Now we can run our project again. And this outputs the dollar rate for us in a slightly more friendly format. Now let's think for a moment about a real-world project. If I have a couple of coworkers that are working with me on this project, how do I make sure that they have the same packages installed? Of course, I can write a README or something similar, but they would still have to install everything by hand. And the solution to this is to create a so-called requirements file. And we can do so by saying pip freeze. This lists the installed packages with their versions, and the common thing to do is to save this in a file. We can do that by running the freeze command and adding a greater-than sign and then the file name requirements.txt. And this creates a file called requirements.txt, let me show you, which contains exactly the list of packages and requirements we saw above. Now let's just pretend we are one of our colleagues. I'm going to get out of this environment and go to my virtualenvs folder and create a new virtualenv for this imaginary colleague. In this case, it doesn't really matter which Python version you use because my script will run on both Python 2 and 3. But in reality, you would probably make sure that everyone in your team runs the same Python version. So your colleague makes his own environment and activates it like this. And remember, of course, that on Windows, you use different slashes and that bin is called Scripts. And let's go back to our project. Of course, our imaginary colleague with his own system cannot run the program because he doesn't have the requirements installed. But all he has to do is to use the now-present requirements file to automatically install all the packages he needs. And he does so by calling pip install -r and passing the requirements file. And this will install exactly the same packages with the same versions that we had installed before so we know that now the program should work. Let's try this. And it does. So you can use the requirements file to make sure that everyone in your team has the same things installed in their virtual environment even though their environments may be on different operating systems or use completely different configurations. All they have to do is pip install -r and pass the requirements file.

Projects, Requirements, and Versions

So how exactly do Python projects relate to virtual environments? Well, a Python project is a folder that contains source code and is probably under version control. The Python source code itself is unaware of the existence of your virtual environment. We've actually already seen that we can run the same project from within various different virtual environments. A virtual environment, on the other hand, is a folder that contains packages, tools like pip, a Python interpreter, and more. And the best practice is to keep your virtual environments separate from your Python projects like in a completely different folder reserved for virtual environments. As you can see in the picture on the left, each of my projects in the dev folder has its own virtualenv in the virtualenvs folder. Of course, I always activate the right virtualenv before working on one of my projects. So usually you will just have one virtualenv for every Python project, but if you want, you can create multiple venvs for a project, for example, if you want to test a project with different Python versions. And if you really want, you can even reuse virtual environments for different projects, although I don't really know why you would do that. So usually each of your colleagues will have their own system and their own virtual environments. When you have installed the packages you need to make your code work, you want to share the list of these packages with your teammates to make sure that each of you has the same things installed. To do this, you create a requirements.txt file with the command pip freeze. And then you use a greater than sign to send the output to a file called requirements.txt. The file that this creates will look something like this. It lists the packages in your environment with their versions. Normally you will put this file in version control and update it every time the dependencies of your project change. Your teammates can then call pip install -r requirements.txt, and this will make pip install all the packages listed in the file into their virtual environments. In requirements.txt, you can not just list a specific version for package, you can also say that you need a version before or after some other version or that you accept any version except a specific one. Pip will install the latest stable version that matches the constraints you give it. The pip install command actually also supports selecting package versions. So here we're selecting version 0.9 of Flask, and in this example, I install any version of Django before 2.0. Please note that the greater and smaller than signs have special meaning on the command line, and that's why we need to put the entire package specification here between quotes. If you have an older version of a package installed, and you need to upgrade it to the latest version, pip can also do this using the command pip install with the -U switch, which is short for upgrade. Note that this has to be a capital U. In a similar way, you can use pip to upgrade itself by saying pip install -U pip. If you're on a Linux system that has a pip

installed by the system package manager, please don't run this sudo because you might mess up your system's pip.

A Real-world Example

So let's have a look at it real-world GitHub project and how we can make it work with virtual environments. So what I'm going to do here is I'm going to start by cloning a project from GitHub. In this case, I'm going to clone Flask, which is a very popular Python framework for writing websites. So after cloning this, I have a directory flask in my home directory, and we can also see what's in there. This is the contents of flask as I just got it from GitHub. Now before we start working with this, of course, I want to create a virtual environment. So let's go into my virtualenvs directory and create a new virtual environment. Very well. Now I also want to activate this, so let's do that. And now I have an active virtual environment, let's go back into the external Flask project. Good. Now the first thing you might notice here is that there's no requirements.txt file here. So why is that? Well, requirements.txt files are usually meant for projects where you have control of your runtime. So basically if you write an application or a website which is going to run on one of your servers, and you're in control of the environment where that's going to run, that means that on that server, you're going to install all the packages from your requirements file. Flask isn't like that. Flask is a project that is meant to be distributed to users through pip. So basically it's like a library that you're going to install as a Python package with pip. In that sense, it's very different from creating a standalone application. Now when you install a package with pip, one of the things it does is it runs setup.py, which we can see here. This is the setup.py for Flask, and we can look at what it contains. So this is a setup script that will run when you say pip install flask. And scrolling down here, we see a lot of information about Flask. And here actually are its requirements. So as you can see, this is very similar to a requirements.txt file, but these are meant to be installed automatically by pip when you do pip install flask. So basically similar requirements but in a different file. Now if I were a Flask developer, and I wanted to, say, do some bug fixes on Flask, I also want to install these requirements. But I also want to be able to run Flask and test it after I do my changes. So in that case, you can install Flask in a special way for a developer, and we call this an editable install. Now before I do so, first, I'm going to move one level up outside of the Flask directory, and then I'm going to say python -m pip install -e flask. And in this case, Flask doesn't refer to the name of the package that's to be found somewhere on the internet, but it refers to the Flask directory that's right here on my disk. So running this, on the second to last line here, you see Running setup.py develop for Flask. And this means that I now have a development installed for Flask. And you see that pip also installed all the requirements for

Flask. But now if we do pip show flask, you see that the package installation now refers to the cloned Flask source code from GitHub. So this is a development install. It's slightly different from an application that you would be developing where you would just use requirements.txt. In this case, we have to do an editable pip install that will take the requirements from setup.py.

Another Example: Testing with Tox

Now there's another thing I want to show you here, which is a very clever way of using virtual environments. Looking at what's inside the Flask directory, there's also a tox.ini. Now tox is a very interesting package that lets you test your code against different Python versions in different virtual environments. And just because I think it's so cool, I want to give you a little demo of how it works. First of all, I'm going to install tox in my current active virtual environment. Next, let's take a short look at tox.ini. And this is the tox configuration file, and you can see that tox is configured to test Flask against several different Python versions. So we have Python 3.6, 3.5, 3.4, and 2.7. Tox is going to try to run the unit tests for Flask in each of these Python versions. So it's going to set up a new virtual environment automatically for each of these versions, install the requirements that it needs, and then run the unit tests. And it even has its own dependency list here. So, again, here is a different requirements list, but this is the list of requirements for testing the project. And to see all of this in practice, all I have to do is go into the Flask directory and call tox. And this tries to set up a virtual environment for each Python version listed in tox.ini. Well, I don't have Python 3.6 install, so that fails. But for Python 3.5, you see it's setting up a virtual environment, installing its dependencies, and now it's happily running unit tests. Once those are finished, you see it setting up an environment for Python 2.7, installing dependencies there, and running unit tests for Python 2.7. So there you have it. This was a little bit more of a real-world example of what you can do with virtual environments and requirements. And that's it for this module. I've given you a little bit of background about why we need to use virtual environments. After that, we learned how to create a virtual environment, and we saw what's actually inside one. And I showed you how to work inside an active virtual environment. I also took some time to talk about defining and sharing the dependencies for your project using a file called requirements.txt. So let's move on to the next module in which I'll talk about virtualenvwrapper, which is a tool that makes working with virtual environments a lot easier.

Using virtualenvwrapper to Make Your Life Easier

Intro and Installation

In this short demo, we'll take a look at virtualenvwrapper, which makes working with virtual environments a lot easier. The next tool I want to talk about is called virtualenvwrapper, and although it's not as much a standard tool like pip or virtualenv, it does make your life so much easier, and it's so popular that I want you to know about it. Virtualenvwrapper is a more user-friendly wrapper around virtualenv, and it allows for easy creation and activation of your virtual environments, it lets you bind your projects to your virtual environments, and as the number of projects you work on grows you'll find that you appreciate it more and more. So let's jump right in and see how to install virtualenvwrapper. I will show you how to use the package for Linux and Mac OS, but there's a very similar package for Windows called virtualenvwrapper-win that works pretty much the same. So on Linux and Mac OS, we install virtualenvwrapper globally just like we did with virtualenv in the previous module. And for Windows, you would add -win here at the end. Now in my system, of course, I don't do that. And to run this, I have to be an administrator, so let's add sudo of the start. Before we can actually use virtualenvwrapper, I need to do some configuration to make sure that it gets loaded when I login. On Windows, this is not necessary at all, so you can skip to the next clip. First, I need to know where exactly virtualenvwrapper was installed. And to do so, I use the command which. Note that I ask for the location of virtualenvwrapper.sh. Don't forget this extension. Now I'm going to copy the output, and now I'm going to open an editor called nano, which is an editor that works inside the terminal, and I'm going to edit a file called .profile. This file is read by your shell whenever you located. You may not have a .profile in your home directory, and in that case, you can simply create a new one. In my case, you see there's already some default code in here. And let's go all the way down to the end, and I'm going to add the following. I call the source command with the location I just copied. And this will cause the shell to read in the virtualenvwrapper code and make its commands available to us. Now to test this, first I have to save this file and exit the editor. And now I have to restart the shell. So I'm going to log out and log in again. And the first time we log in to the system, you now see that the virtualenv creates a bunch of scripts in a folder called .virtualenvs in my home directory. This is also where it will expect any virtual environments to be located. Now in my situation, I've been using a different place to store my environments, and that is virtualenvs without a period dot. Now we can actually configure the location of our environments, and to do

so, I will add another line to .profile. Let's start the editor again. And going down, here I'm going to add the following line, export WORKON_HOME= and then a path to the virtualenvs folder in my home directory. Please make sure that you don't use spaces around the equal sign here or this will not work at all. Also, I added quotes around the path because this way if your location contains spaces or other special characters, it will still work. Now, of course, you can set this to any location you like, or you can just not use it at all and just use the default .virtualenvs. Anyway, the nice thing is that with this set up, virtualenvwrapper will pick up my existing projects. Now to make this last change go into effect, again, I have to restart the shell, and now we're ready to start working.

Working with virtualenvwrapper

So let's do some real work and see how to use virtualenvwrapper. By the way, all of this should work on Windows if you installed virtualenvwrapper-win. We'll see how to activate our projects, switch between them, create and remove projects, and also how to link projects with their virtual environments. The first thing we can do now that we have a working virtualenvwrapper is call the command workon, and this lists the environments in my virtualenvs folder so they are detected automatically, and I can activate any of them by saying workon followed by the name of an environment, let's say rates_py3. And we see the environment being activated. Now saying pip list shows me that Requests and Box are installed in this environment like we did in the last module. So this is actually the same environment we've been working on before. I can easily switch to another environment by saying workon flask, and now I'm in the other environment. So that's already quite cool. It will save us a lot of typing. But there's more. I can create an entirely new project by saying mkproject followed by the name of a new project, but here I get an error because for this I have to set up the location of my projects. To do this, again, I need to go into my editor. And I'm going to add another line setting up an environment variable PROJECT_HOME. On Windows, you can set an environment variable with the same name by going into the properties of your PC and then the Advanced System Settings. We've seen this in the demo about installing pip. So here I'm telling virtualenvwrapper that my projects will be in home/reindert/dev. Like before, please make sure that you don't use spaces around the equal sign here. Now let's save and exit the editor. And, of course, I need to make sure that this folder actually exists. So let me create it. And to lock the new configuration, I'm going to log out and log in again. Very well! Let's try to create a project again. Now this creates both a directory for my project, so for my actual source code in home/reindert/dev/new_project, as well as a virtual environment in home/reindert/virtualenvs/new_project. It also immediately activates the

environment. Isn't that great? Now let me show you something more. Let's say I switch to another project, flask, and I'm going to move into its folder. So let's say I'm happily working on my flask project, and at some point I want to switch back to the project I just made. I type workon new_project, and not only does this activate the right environment, it also switches into the project directory. So if you created a project with mkproject, this works automatically because the project and environment are bound when they're created. For my older project, this doesn't work because they were not made this way. But I can still bind these older environments to project location. To do this, first, I activate the environment, and note that we now don't jump to the flask project yet. So I'm going to move there, and now I can bind my active environment, flask, to this folder by saying setvirtualenvproject. So this prints Setting project for flask to /home/reindert/flask. So now if I switch back and forth, let's switch to the new-project again, and back to flask. Now we automatically move to the right location in the file system to work on our code.

Review

To install virtualenvwrapper on Mac OS or Linux, you call pip install virtualenvwrapper, but you should be aware that you may need to use sudo. After installing, you need to add a line of configuration to a file called .profile in your home directory, and this line is source followed by the location of the virtualenvwrapper.sh script. Now make sure to check the location of the installed script by calling the which command as I've shown in the demo. After doing this, make sure to restart your shell. And the default location for your environments will be in a folder called .virtualenvs in your home folder. The normal virtualenvwrapper script doesn't support Windows, so if you use Windows, you should install virtualenvwrapper-win instead. In this case, you don't need any extra setup step. And the default location for environments will be in a folder called Envs under your USERPROFILE directory. Once virtualenvwrapper is installed, it allows you to list all your current environments with the workon command. Or you can call workon with the name of a project, which will switch to that project and activate the environment with the same command. To leave a virtual environment, you can use the familiar deactivate command. The mkproject command creates an empty project, as well as a virtualenv, and binds them to each other in a single step. And this means that whenever you start working on a project, you will move into its folder automatically with the right environment activated. Like the virtualenv command, you can use the -p switch to start a project with a specific version of Python. Of course, that Python version has to be installed on your system. For projects created without virtualenvwrapper, there's the setvirtualenvproject command, which will bind an existing project,

like one that you just checked out from version control, for example, to a virtual environment. Make sure that the actual virtual environment is active and that you are in the project directory that you want to bind. There are also several commands for managing virtual environments that I didn't show in the demo. With mkvirtualenv, you can create a new virtual environments, rmvirtualenv will remove an environment, and the mktmpenv command will create a new nameless virtual environment for a single use, which can be really nice if you just want a quick environment to test stuff out in. You can configure the location of your projects and environments with two environment variables. The names of these variables are the same whether you're on Windows or any other system. Now on Linux and Mac, you add the lines shown here to your .profile, and please make sure not to put any spaces around the equal signs. The WORKON_HOME variable specifies the location of your virtual environments, and the PROJECT_HOME is where your projects live. By the way, if you want to make project commands to work, you need to set PROJECT_HOME or you will get an error. On Windows, you don't set these variables in a file. But, instead, you go to your Advanced System Settings like we've seen before when installing pip. And that's all I have to say about virtualenvwrapper. It's really a great tool that makes working with projects and environments a lot easier. We've seen how it helps us to create and switch between projects and how to configure it. I hope you will enjoy using this great tool just as much as I do. Now let's go on to the next module and learn about another great tool called pipenv.

Choosing the Right Tools

Introducing pipenv and poetry

Hi, and welcome to this final module. I'll talk a bit about current developments in the Python packaging world and also just some other tools that we haven't seen yet. So far in this course, you could argue that I've been quite conservative in the things that I have shown you. You see, although pip and virtualenv and to a lesser extent virtualenvwrapper are very much standard tools that you should really know, we live in a rapidly changing world. For many different reasons, people haven't been satisfied with the way package management works in Python, and ever since the early 2000s, efforts have been underway to implement better tools. Unfortunately, the result at the moment is that it's pretty much one big mess. For almost every task, there are multiple conflicting software projects with different philosophies and workflows. In this module, I will show

you some of the more popular new tools you might want to use yourself. That being said, I cannot foresee the future, and I cannot tell you which way the Python world will go. That also means that if you're going to adopt a certain cutting-edge tool, that tool may or may not become a standard, and your skills with that new tool may not be future-proof. So, as I said, the situation in the Python packaging world is quite messy. To improve this, the Python Packaging Authority was created, which is a working group that maintains most important packaging projects with as its mission to provide a relatively easy to use software distribution infrastructure that's also fast, reliable, and reasonably secure. It has a site at pypa.io, which contains lots of information, a wonderful user guide, and a list of software projects around Python packaging. This site is a very good starting point if you want to know more. We started the course by talking about installing packages, and the standard tool for this is pip, which installs packages in a format called wheels or sometimes in an older inferior format called eggs. There's also an older Python installer called easy_install, but everything it does, pip does better. So please don't use it. And if you see it mentioned in some documentation, you should be aware that this is an older tool, and pip is preferred now. And then there's the part that you will probably spend most of your time at, development. Of course, this will mostly mean working with code. But in the context of this course, I'm talking about dependency management, and so far we've seen a little bit of this with the requirements.txt file, which holds a list of project dependencies that are installed by pip. To isolate the dependencies from those of other projects, we use virtual environments for which there are two main tools, virtualenv and venv, which is installed with newer Python versions. Of course, we also saw virtualenvwrapper, but that's more like a utility for convenience, so I'm going to leave that out of this slide. So regarding these two tasks here, dependency management and project isolation, this is where currently there are a lot of interesting things happening. There's a lot of effort going on to improve the situation here because lots of people are not satisfied with the way virtual environments, requirements, and pip work together. And this is what I want to focus on right now. One project that has got a lot of attention recently is pipenv, which aims to combine the functionality of pip and virtual environments in a single tool, and it also improves the way dependency management works. There's a similar project called poetry with more or less similar goals, which also looks very promising, and those are just the two major ones. At this point, neither pipenv nor poetry is a standard tool for the Python community, and it's unclear whether either of them will ever replace pip and virtualenv. So there's no actual need to know these tools. You can be a productive Python programmer without them. But you might come across projects that do use these tools. So at the very least, you should be aware of their existence and of the fact that there's some movement in the Python world towards a different approach for project and dependency management. Now there's one other project that I really

have to mention here that's quite unlike all the others, and it's called Anaconda. This is basically all the other tools combined and more. The people at Anaconda distribute their own Python, have their own package management, and their own distribution system. They have a tool called conda that does everything pip and virtualenv do and much, much more. You can also install non-Python packages with conda, let's say an editor or a database like Postgres. It even knows about pip and the packages it installs, so you can use both at the same time. It's really quite amazing. Anaconda seems to be mostly aimed at the data science community, and the default install comes with a huge amount of pre-installed packages that are awesome if you are into data science and analysis and probably just a lot of overkill if you aren't. That being said, as much as I like Anaconda, it really isn't part of the standard Python ecosystem. So for this course, this is all I will say about it. I do encourage you to try it out sometime if only to see how Anaconda approaches all of the things we've seen in this course. Before we go and take a closer look at pipenv and poetry, let's talk about project requirements. We're already familiar with requirements.txt, which is a text file that we can use with pip. So it's more or less a standard. It's a popular way to store your project dependencies, but it has some limitations. For example, there's a problem of dependency resolution. Suppose we have two dependencies, A and B, and they both depend on the same package C. But they have different requirements for the version of package C. Let's say one needs C to be newer than version 1, but the other wants C to be older than version 2. What version of C will be installed? Well that probably depends on the order in which you install A and B, or maybe even on the exact moment when you install things. In other words, when dealing with a dependency tree where your dependencies have other dependencies, the outcome isn't deterministic. If two people have the same requirements.txt, that doesn't at all mean that they will have exactly the same packages installed, that is, unless you list every sub-dependency of every dependency in your requirements file. If your project becomes more complex and adds more dependencies, this will get ugly. Now there are other reasons why people dislike using requirements.txt, but this is probably the most important dealbreaker. But there are some alternatives. Pipenv, one of the new tools I will show you in a moment, uses its own file format called pipfile. It makes sure that the set of packages installed for your project is repeatable and deterministic. So you will always be sure that everyone on your team has exactly the same set of packages installed. You can also use this for deployment if your project is the kind of application that you deploy somewhere where you can use pipenv to install its dependencies. Not everybody agrees with the approach that pipenv takes, and there's another format called pyproject.toml, which is a text-based format a bit like the ini files you may know from Windows. And one of the tools that uses it is called poetry. The pyproject format is actually standard for specifying project

requirements. Like with pipfile, we can use this to make our insulation repeatable and deterministic. So let's go and have a look at the cool new tools, pipenv and poetry.

Pipenv

Now let's look at what these tools look like. We'll take a quick look at pipenv and how we can create a project, install dependencies, and run some code. I just want to give you an impression of the user experience of this tool. Let's give pipenv a try. Of course, we need to start by installing it, and we can do this with pip. Just like we did with virtualenv, we have to install this globally. So on my system, I need to use sudo. Very well. Now this is really all I need to do. I'm going to move into my demos directory, and here's the same script we saw before. And this script needs Requests and Box to run, so let's use pipenv to install those. And pipenv now does multiple things. It creates a virtual environment, which just like with virtualenvwrapper is bound to this specific project, it installs my dependencies in the virtual environment, and it creates Pipfile and Pipfile.lock. So now we have three files in this folder. And looking at the contents of the Pipfile, this file contains our dependencies, requests, and python-box here in the packages section. And because I haven't specified any version when installing, here it says that basically any version will do. There's also a dev-packages section for development-only dependencies. So here I could put my code checker or unit test runner. The project's Python version is also in here, as well as the source where the packages are downloaded, which, by default, is PyPI. So this is somewhat like a requirements.txt file, but it has some more functionality. You can think of this as a general human readable specification of dependencies. Looking at the other file, Pipfile.lock, this is a computer-generated file that stores the exact version of every single package I installed. And this is what pipenv uses to make the dependency tree deterministic. If I put Pipfile and Pipfile.lock in version control, then my teammates will have exactly the same version of every single package installed. If at some point you need to upgrade packages or install other packages, you can always regenerate Pipfile.lock. Right now I'm not in an active virtual environment, and that means that I cannot run my program. To do that, I can use the pipenv run command like this, pipenv run python, and then my scripts name. So pipenv run will run any command you give it inside the virtual environment. Something else you can do is to run pipenv shell. This starts a new shell inside the active environment. So now I can run my program simply by saying python get_rates.py. Now to get out of the environment in this case, I have to take care to say exit instead of deactivate. Now what if I want to work with another Python version. I can do this with the switch --three. If I say pipenv install --three, this tells pipenv to install everything from the pip file in a Python 3 environment. We see that pipenv automatically destroys the old virtual

environment and starts a new one with Python 3. It also installs my dependencies so we can immediately run our code. Note that pipenv now complains because our Pipfile still has the line that it requires Python 2.7, and we're now using Python 3. So in this case, I should edit my Pipfile and correct the Python version. Very well. That gives you a very short overview of what it looks like to work with pipenv. It really combines functionality from pip, virtualenv, and virtualenvwrapper, and even add some extras. It's a popular new tool, and if you find it easier to have one single tool instead of three different ones, this might be something for you.

Poetry

So let's check out another tool called poetry. And we'll do the same things, start a project, install some dependencies, and run our code. Here we are at the poetry home page. And if I click Documentation here at the top, here are the installation instructions. And there's a line of code here that I can use to install poetry. So I'm just going to copy/paste that into my terminal. And what you immediately notice is that in my opinion, at least, poetry really has a friendly user interface. It immediately gives some info about what the installer does, and it added some code in my .profile. So I don't even have to do that myself. One way to activate this change is, of course, to log out and to log in again. But there's another way, which is to run the command shown here. So let me just copy/paste that. Very well. And now I can start working with poetry. Let's start a new project. I'm going to call it currencies. This creates a new folder. Let's take a look. And poetry has generated a whole project skeleton for me. There's an empty README file, a currencies folder, which is actually a Python package with an empty init file in there, so that's meant to hold the code for my project, a folder for unit tests, and the pyproject.toml, which is where my dependencies will go. But let's start by installing our requirements, and slightly surprisingly, I have to use the command poetry add instead of install. Apart from that, it works pretty straightforward. And, again, we see that the output is very user-friendly. Now let me just copy our familiar script into the package folder. And to run our code, we use the same approach as with pipenv, which is to say poetry run python. But the difference is that the location of the script is now slightly different. Or, again, we can start a shell with an active environment. And because in this case we actually have a package, I can start Python and import the module from the package. And this, again, runs our code. Good. Let me exit Python and then exit the shell, again, not using deactivate, but the exit command. And let's take a look at the dependencies file, pyproject.toml. And this is actually quite similar to the pip file. We see a section for dependencies and one for development time dependencies, and there's a separate tool.poetry section here which lets you fill in some metadata about your project. There's also a build-system section, which is there

because poetry will actually create a package for you that you can upload to PyPI. That's something that pipenv doesn't do. There's also a poetry.lock file, again, to make the build deterministic and repeatable. Now creating a project for a different Python version is not as easy with poetry. The best way to make this work is with a separate tool called pyenv, and that's beyond the scope of this course. So that should give you a little bit of an impression about pipenv and poetry. I really think they are quite similar, although they have different ideas about things like what a project is and how to do packaging.

Conclusion

Pip and virtualenv are still very popular, and they're not going anywhere soon, so the safe choice is just to stay with these familiar tools, and you'll be fine. If you prefer to be a bit more on the cutting edge, and you like a more user-friendly workflow, either pipenv or poetry might be a good choice for you. Pipenv at the moment seems to be slightly more popular, but poetry seems to be more friendly to use and supports packaging in case you need it. If you're into data science and analytics, Anaconda is probably your best choice. And that's it for this course. In this last module, I've tried to give you an overview of the current situation regarding Python dependency and project management, but what I hope you will take away is that there are many different approaches, and the situation is changing. You can always decide to just use pip and virtualenv and, if you like, virtualenvwrapper. But we've also seen a glimpse of the future with both pipenv and poetry, which offer a much more integrated user experience. That's it. I hope you enjoyed watching this course and that you learned something that will help you go forward on your path, whichever tool you might choose. Happy coding, and thanks for watching. This was Reindert-Jan Ekker for Pluralsight.

Course author

Reindert-Jan Ekker

After studying Artificial Intelligence, Reindert-Jan has worked in many different roles in the software development life cycle, from tester and developer to Scrum Master. Then he decided to go...

Course info

Level	Beginner
Rating	★★★★½ (53)
My rating	★★★★★
Duration	1h 23m
Released	2 Feb 2019

Share course