A newer version of this course has been released. View now

The Python Developer's Toolkit

by Reindert-Jan Ekker

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Learnin

Introduction

Introduction

Hi, my name is Reindert-Jan Ekker and welcome to this course called The Python Developer's Toolkit. In this course we'll explore a number of tools that are likely used in the Python community and that every Python developer should know about. In this short introductory module I will go over two things, what can you expect from this course and what you should already know to be able to follow the course. The basic idea behind this course is to introduce you to a number of tools for tasks that occur in just about every real-world Python project, which by extension means that every serious Python developer should know these tools. I'm going to stick to standard tools. Most of the tools we'll learn about are either part of the standard Python installation or when they're not, most of time they're the most popular tool for that particular task and used by most large projects. And in some cases these tools are set to become the future official Python standards. So why is all of this important? Well, as a Python programmer you're not just using the Python programming language and interpreter, you're also using the standard library. And once your projects start to grow and become more complex you'll find that there's even more beyond that. There's a whole ecosystem of tools and best practices that you should learn about. And this course aims to help you along the way to becoming a well-rounded Python professional.

Intended Audience

Now the best moment to learn about these topics would be, in my opinion, when you already have written some simple scripts and projects and you feel ready to move on to more serious real-world projects. And that means you should at least be somewhat familiar with the Python language. I'm not doing a lot of complicated coding in the course so you don't really have to be a Python guru. One thing you should at least understand is how to import a module form the standard library. Now, all the tools I will show you run from the command line. We're not going to use any graphical user interfaces. Basically that's because in the open source world, command line tools are still preferred. And I do agree with that because I think they give you a lot more power than point-and-click interfaces. Now if you don't know about the command line, don't worry, you'll be fine. Basically, you won't need to know anything about the command line, you can simply type the commands like I show them to you. But you might want to consider learning about the command line as well since that's a real power tool, and it will make you a lot more productive.

Course Overview

So what's in the course? We're going to look at the following six topics. First, we'll see how to find, install, and uninstall third party packages with a tool called pip. So that will basically allow you to do something called dependency management. From there we'll move on to setting up an isolated development environment with a tool called Virtualenv. Once you start working on multiple Python projects on one system, this is the tool you really can't do without. And in my opinion these two first topics, pip and Virtualenv, are really tools you must know, and you can't do without at all as a Python developer. Then we'll move on and learn about a tool called Pylint that you can use to automatically check your code quality. From there we'll move on to using the Python debugger, which is a standard tool that comes with Python that will help you solve errors in your code. We'll see how to generate beautiful code documentation with a tool called Sphinx. And finally we'll see how to package your Python projects and share it with the world. And that's it for the intro. I hope you enjoy watching this course.

Managing Python Packages

Introduction

Hi, my name is Reindert-Jan Ekker and welcome to this module where we'll see how to manage Python packages. So this module is centered around Python packages and we'll see how to find packages in a central repository, and, of course, how to install and remove them. I'm also going to tell you a bit about all the tools that are available to you. Actually, there's quite a few and it can be overwhelming when you're new to all of this. So I'll tell you what the recommended tools are and which ones you should avoid. Now you should be aware that after watching this module, you don't actually know everything there is to know about packaging. Normally you'll want to set up your own separate environment for each project and install packages in there. So to get the complete picture be sure to watch the next module as well.

Demo: Imports and Packages

Now let me give you a little background information about packages. I installed a package called requests, which I can import simply like this. And the request library allows me to use HTTP in a very simple way. So for example, this statement will retrieve a random Chuck Norris joke. So that's very nice, but what exactly happens when I run the import statement above? Where is my package installed and how is it found? Well, there's a variable I can inspect to find out. First I'll have to import a library called sys, this is part of the Python standard library. And now we can ask for the contents of sys. path. And 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. Now this is a very long list and it will vary between installations. And I want to focus on only two elements of this list, namely the first and the last. The first element here is the empty string, and that denotes the current working directory, that's the directory from where the Python interpreter was started. So when I said import requests, Python started with searching for a package called requests in the current working directory. And then when it didn't find anything called requests in there, it searched the second directory on the list, and then the third one, etc, etc. And then at the end, the very last item on the list, is a directory called site-packages. And in my case, that's in usr/local/lib/python2. 7, and the reason that it's in usr/local/lib is because of the way I installed Python on this machine. You can also see before that another directory called Library/Python/2. 7, and that's actually the system default on Mac OS. But then I installed my own Python version in a different location and this was added to the list. Now the site-packages directory is where we will install third party packages like requests. So of course, I'll expect the requests library to be in this directory. Let's check. I'm going to Copy this, and exit Python, let's go into that directory, and see what's in there, and as you can see there's a lot of stuff in here, but one of the things that's in here is indeed the requests package. So this is where the interpreter finds, and loads the request library from.

How an Installed Package is Found

So we saw that when we import a Python module the interpreter searches all directories in the system, path variable for installed packages. There can be any number of directories in that list, but generally the first one is the current working directory from which the interpreter has been started. Of course, when you're working on a Python project this enables you to import names from the other Python files in your projects. And generally at the end of the list we find a directory called site-packages. Now, this is where third-party packages should be installed. Where this directory is located depends on your operating system, your Python version, and of course, where you installed Python. But it should be inside your Python installation. Now you should also be aware that this directory might not even be called site-packages at all. For example, under Debian, it's all dist-packages.

Python Packaging: Current State of Affairs

So now that we know all of this, what's the best practice for managing packages? How do you find packages, download them, and install them in a reliable way? Well it turns out this question is not very easy to answer. If you try Google you'll find lots of contradicting and outdated information. There's a number of tools called distutils, setuptools, and more. And there are even multiple different package formats. It's a very confusing situation and it's easy to be fooled and use an old deprecated toolset by mistake. So to clean up this mess a new working group was created called the Python Packaging Authority. It has set a roadmap, and gives recommendations for what tools to use. Currently, its advice is as follows: If you're writing a package and want to distribute it, use setuptools. Don't use distutils because that project has been discontinued and merged with setuptools. And also don't use Easy_install, which is now included with setuptools as well. Now I'm not going to explain how to create your own package with setuptools right now, we'll see more about that in the last module of this course. For now let's focus on the use case of using and installing someone else's package. And for that you should use a tool called pip, that's the one we'll focus on for the rest of this module. Now like I said before, be sure to watch the next module as well because that covers virtualenv, and you really want to use pip in combination with that.

Installing Pip

So the Python community is working towards a standardized packaging mechanism, and as part of that it has been decided that pip should come pre-installed, and this is the case since Python 3. 4. So if you are using Python 3. 4 or later, you won't need to install pip because it comes pre-installed. If, on the other hand, you're using Python 2 or an earlier Python 3 version, you can go to pipinstaller. org and follow the instructions there. Basically it will tell you to download a script called get_pip. py, and run it, that will install pip for you. Now I'll show you this in a short demo in a moment, and the first clip we'll show you how to install pip on Linux and Mac OS, and if you're a Windows user you can skip to the clip after that where I'll show you how to do this on Windows. Now note, that Linux and Mac OS users will probably need to run the get_pip script with sudo. We'll see this in a demo as well. Now for Mac users who use Homebrew, by the way which is excellent and I really recommend, so people who use Homebrew to install software, Homebrew installs pip with Python by default so you won't have to install pip separately because it comes with Python if you install it with Homebrew. Now there's also a note for Windows users, when you run and get pip, you will find pip in the Scripts directory of your Python installation. Now you can make calling pip easier for yourself by adding the Scripts directory to your PATH.

Demo: Installing Pip on Linux or Mac OS

So here I am, logged into a brand new Ubuntu installation on a virtual machine, Python is already installed, now let's see how to install pip. By the way, if you're installing on a Mac, the procedure will be the same so you can watch this video just as well. Now I already downloaded the get-pip script into my downloads directory so let's move in there, and here's the script. To run it I say Python get-pip. py. Now this command will run the script and it's the same regardless of whether you're running Linux or Mac. But we're going to install pip in a system directory and for that we need administrator privileges. So we need to prefix this command with sudo, this will run the command python get-pip. py with admin privileges. So let's press Enter, and now that pip is installed we can use pip to install other packages like for example, requests. But as you can see here I get a Permission denied error and that's apparently because I don't have permissions to write in the dist-packages folder. So as a normal user I don't really have the permissions to install a new Python package in a system-wide manner. So again, here I have to prefix my command with sudo. By the way, in the next module we'll see a way to install packages on a per user, per project-basis, and you won't need to use sudo anymore. But for now let's just do it this way, install requests with sudo, and now we can run our little test again, let's start Python, and import requests, and let's ask for a random Chuck Norris joke. Now let's also take a short look at sys.

path again. And this you can see on this Ubuntu machine the list of directories is a lot shorter and the last directory, as we already saw above, is called dist-packages and not site-packages. But apart from that it works exactly the same as on a Mac or a Windows machine. So let's check out the same procedure on Windows.

Demo: Installing Pip on Windows

Installing pip on Windows is very similar to installing it on a Mac OS or Linux. Basically you go to pipinstaller. org, and you go to the Installation page, and then here you see to install or upgrade pip, securely download get-pip. py. And I'm going to say Save target as, and download it, let's say, to my Desktop. Very well. Now all I have to do is click Open and script will run. And as you can see it downloads and unpacks pip for me. Now before I show you that it works, we'll have to check something because on Windows it's very important that you put the Python installation on your PATH to be able to run Python and installed Python scripts like pip very easily for the command line. Now on this Windows system I installed Python 2. 7 and I know that the installer for Python 2. 7 doesn't automatically add the Python installation to my path. I'll right-click on This PC, and go to Properties, and then Advanced system settings, and then Environment Variables. Very well. Now down here you can find the Path setting, let's click Edit, and if I inspect the path I do not find any reference to the Python installation here. So now I'm going to copy the Python installation path, very easily here by saying Copy, and I'm going to Paste that in here, and then use a semicolon as a separator, and very importantly, I want to add the Scripts directory as well, because that's where pip is. So I'm going to add that as well, again add a semicolon, then I'm going to click OK. Say OK here, okay, and now on the command line I should be able to call the pip command, and as you can see it works. Let's install requests, very well, seems that works as well. Let's call Python, import requests. That works. And for a moment let's look at the sys. path, and here you can see again, the first element here is my current working directory, and last element is the lib directory in my Python installation followed by site-packages. So basically that's all you have to do on Windows to install pip.

Installing Packages With Pip

So we just saw that to install a package you simply call pip install with the name o your package. Pip will download and install the package and any other packages it might depend on. Sometimes you will need a specific version of a package and you can use this syntax to install that specific version. So in the example we install request version 1. 0. You can also use comparison operators like this, where we say that we need a request version of 2. 1 or greater. Similarly you can use the smaller than operator as well. Now notice that there's quotes around the comparison in this last example. Don't forget to quote the comparison when you call pip like this, because the larger and smaller than characters are interpreted by most shells as input output redirection, and that's not what you want. Now there's another very important command, and that's pip uninstall and that will let you remove a package. These are the core commands, the ones you will probably be using a lot. But I also have a little warning for you, don't go ahead and install a lot of packages right now. Using pip like this, on its own, will install packages system wide, and you really don't want that because it will cause conflicts. So feel free to play around a little and try these commands, but make sure to clean up after yourself to uninstall the packages that you install. And watch the next module where I'll tell you about virtualenv, and how to use pip in combination with that, and then you'll know how to properly install packages.

Inspecting Packages With Pip

Let me just give you a very short overview of three pip commands you can use to inspect packages. First of all there's the pip list command, which will show you a list of all installed packages. It'll simply show you the name and version for each package. You can also ask for a little more info on a package with pip show, which will also list the installation location and requirements for that package. But most importantly you can search for a package. This will search all package names and descriptions for the query you supply. Now you'll probably wonder where all these packages that pip searches are located. And when you install something where is it downloaded from? Well there's a central repository called pypi or the Python Package Index, or more informally, the Cheese Shop. That's a name based on the famous Monty Python sketch. Now instead of using pip to search for a package you can also visit the pypi site to look for packages. Now let's see some of this in a short demo.

Demo: The Cheese Shop

So I'm back on the prompt of my Ubuntu virtual machine, and let's do a pip list to see what the output's like. And as you can see on this Ubuntu system there's a lot of packages installed by default. The only one I installed myself is called requests, and as you can see its version 2. 2. 1. Now let's ask for a little more information about let's say, argparse, because I'm not sure what that does. Well, apart from the version and installed location that doesn't tell us very much. So let's head for the Cheese Shop and see if we can find out a little more. So this is the Python

Package Index at pypi. python. org, and let's search for argparse. Here it is, let's go there, and apparently it's a module that makes it easy to write user-friendly command line interfaces. So on this site you can find all kinds of Python packages. Suppose I want to connect with Twitter to show Twitter messages in the application I'm writing, I might search for Twitter, like this, and here as you can see there's an actual Twitter package and that will allow me to connect with the Twitter API. So now if I would say pip install twitter, pip would download and install the Twitter package from pypi and put it in my dist-packages directory.

Requirements Files

Now there's one last thing pip has to offer that I want to tell you about, the command is pip freeze. And what it does is it simply prints a list of all currently installed packages and their versions. It does so in a format that pip itself can parse again later. And the normal use of this command, as you see here in the example, is to use a larger than sign to redirect this output into a file called requirements. txt. So that means this command saves a list of all your currently installed packages, in other words all the packages your project depends on in requirements. txt. It will also include the specific version for each of these. Now why would you want to do this? Well suppose you're working in a team and there's a new developer on the team. They want to get up and running quickly so you tell them to check out the project source and then use the following commands: pip install -r requirements. txt. This will go through all the packages in a file and install them with a single command. So after issuing this single command, your new developer has all the requirements for your project installed. Now don't confuse this list of requirements with the dependencies that people download when installing a package, because when pip installs a package with the pip install and then the package name commands, it will re-determine the dependencies to be installed for that package in another way. We'll talk more about that later. Pip freeze simply gives you a list of currently installed packages, which is helpful for developers who want to get their project environment installed quickly. So basically requirements. txt is a development time requirements list, but there's also a completely different mechanism where pip can determine the dependencies for a package on install time for an end user. So when an end user installs a package with pip install, the file requirements. txt does not come into the picture.

Demo: Requirements Files

So here I am in a directory called HelloWeb and basically that's a very simple example project that shows a Hello World web page using the Flask microframework. Currently all it contains is a little Python code and a requirements. txt file. Let's look at what's in requirements. txt. Currently the only dependency I have is Flask. Now before installing this let's look at the currently installed packages. So it shows pip, setuptools, and wsgiref. These three packages are usually always present. So I didn't install them, they're there already after setting up. So now let's install what's in the requirements file. So, as you can see, this installed multiple things. It started with Flask, but Flask itself has several dependencies as well, and so pip downloaded and installed these as well. So among other things it installed Jinja2, and itsdangerous, and markupsafe. So now if we say pip list, we can see that all of these packages are present. And that means that now we can run our little program by saying python helloweb. py. Now we can check out the simple page it generates by going to localhost port 5000 and there is says Hello, Web! That's basically all this application does, but of course this example is about the requirements file, not about the functionality of the code. So let's quit this. Now of course you know that normally a requirements file is generated with the pip freeze command, but when I run pip freeze you actually see a lot of different packages. You see all the dependencies for Flask, as well as Flask itself. Now, of course, one of the things you could do is list all of these in your requirements file, but you're free to edit the requirements file yourself. So what I did, in this case, is instead of listing all the Flask requirements I only picked out Flask itself. And I even removed the specific version, and that's something you have to decide for every project yourself. You see, the code I have in this project is so simple, it'll probably keep working with a lot of new versions of Flask to come. So I don't see a reason, right now, to add a specific version here of Flask in my requirements file, but sometimes you have a more complex project and you start to depend heavily on a specific version of one of your libraries. And in that case, it would be wise to add a line like this in your requirements file. So basically, what I'm saying here is that you can use pip freeze and save it into requirements. txt like this, but you don't have to. You're free to edit it like I just did and basically only list the actual direct dependencies of your code, and you can also choose whether you want to specify the specific versions you depend on or not.

Resources

Now as I said before, you should be careful when searching the web for information about Python packaging since there's a lot of old and contradictory information around. Here are some of the more trustworthy links. Please note that I included shortened links as well, they'll be easier to copy and type into your browser. First of all, there's the official Python Packaging User Guide with the road map and recommendations from the Python Packaging Authority. Here's a link to the pip documentation in case you want to know a little more about pip's advanced options and configuration. And then there's of course, the Cheese Shop.

Summary

And that concludes this module. We've seen what installing a Python package really means, and how to find, query, install, and remove packages with pip. We've also seen how to install pip on your system, and how to freeze your currently installed packages into a requirements file. So let's move on now to the next module in which I'll show you how to move from installing packages for your whole system to installing them independently for each project you're working on.

Isolated Development Environments With Virtualenv

Introduction

Hi, I'm Reindert-Jan Ekker and in this module I'll tell you how to create virtual Python environments for your projects. So I'll start with giving you a little background, explaining why virtualenv is necessary and what it does. And of course, we'll see how to create and remove virtual environments and how to use these environments when working on your projects. I'll also touch on some more advanced options for virtualenv. And finally we'll take a look at a wrapper around virtualenv that makes it a little more usable, called virtualenvwrapper.

Why we Need Virtualenv

In the previous module we saw how to install third-party Python packages with pip, and we installed everything in a system-wide fashion. Now there's 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 some system scripts depend on a library like argparse, which we saw in a demo, now these libraries are installed by your system installer, for example appget, and installing our own packages for the whole system may mess that up. Or maybe you're on a multi-user system where you don't have root permissions to install anything system-wide. Or what about a project which you need to test against several different Python versions, or against various versions of some other library? Well fortunately there's a solution for all of our problems, and it's called virtualenv. It allows us to create Python environments that include the dependencies for your specific project in such a way that they're completely isolated from the dependencies for other Python projects, or any system-wide package installations. Now it's good practice to create such an environment every time you start a new project. You can then install all the dependencies for that project inside the virtualenv environment so they won't cause conflicts with other projects. And that basically means that you should never install any Python packages on a system-wide scale, ever again. You will save yourself lots of trouble by making a habit of always working inside a virtualenv environment. Now if you do so, projects with conflicting dependencies can coexist peacefully.

Demo: Start Using Virtualenv

So we're back on the command line and the first steps to start working with virtualenv shouldn't surprise you, we'll simply install it with pip. Now note that on Linux and Mac OS again, we need to use sudo because it's a system-wide installation of the virtualenv program. From here on we won't need to use sudo anymore. So now virtualenv is installed, we can start working. Now let's start a new project. Before I start the project itself, I'm going to make a virtual environment. And before I make my first virtual environment I'm going to make a new directory where all my virtual environments will end up. Now most people put their virtual environments in a directory called. virtualenvs, so let's follow that practice. And let's move in there. Very well. Now I can simply create a new virtualenv for my project. I'm going to call my project amaze, and my virtual environment will get the same name. So this calls the virtualenv command with the name of my new project, and when I run it, it creates a new directory with that name called amaze. In there it put several things, for example, a bin folder with, among other things, a Python interpreter. It also installs setuptools and pip locally inside the environment. Now, we'll examine the content of these directories in more detail in a moment, but first let's see how to use it. To do so I have to activate the environment like this. Here, the dot means we want to import some shell script code from a file, and that file is inside my amaze virtualenv, and then inside the bin folder, and then it's a file called activate. Now I can run it like this, and what happens is we see the prompt change, it now starts with the name of my new active virtual environment inside parenthesis. So this means we're inside an active virtualenv.

Demo: An Active Virtual Environment

So we're inside an activated environment and I can install packages now. For now let's just install something called the Pylint code checker. We'll see more about this package in the next module so I won't go into the functionality of the package. Basically it provides a script that checks our code for errors. Now this package will end up installed inside the virtual environment, so it won't affect anything outside of there. In other words, any Python projects that live outside this virtual environment will not be able to see this package. The install is totally isolated from the rest of the system. Also note that we didn't need to use sudo to run this because we're only affecting folders that are owned by me. Now let's check out the contents of the virtual environment. Well here we are in my. virtualenvs folder, and here's the amaze virtualenv folder I just made, and if I open that we see three things: bin, include, and lib. Now, let's look at bin. It contains a lot of executables, and among them are Python, the Python interpreter, pip, and Pylint as well which I just installed. Now one of the things activating a virtualenv does is it changes your path, which means that now if I run Python, for example, it will pick up this specific executable inside the bin folder, inside my virtual environment. So it won't run the system-wide installation of Python. Now on a UNIX system I can check that with the which command. I can run which python and you see that this, that the command Python now points to the Python interpreter inside my bin folder, inside amaze. And we can now also run the Pylint script I just installed. And this will try to find errors in our Python code, but of course it won't do anything because there's no code in here right now, so it gives an error. But what I want to demonstrate right now is that I can deactivate the virtual environment, and now we're outside the environment, and now I can't run Pylint anymore. It says command not found, and that's because, as I said, Pylint has been installed inside an isolated environment, and we can't see it from outside.

Review: Virtual Environments

So to create a new virtual environment you use the command virtualenv followed by the name of your projects. Now for Windows users, if virtualenv is not on your path, you can find it in the Scripts directory under your Python installation. And of course, it's always a good idea to add that to your path. Now after creating an environment you have to activate it. On Linux and Mac OS you use the dot command followed by the name of your virtual environment, which, in this example, is myproject/bin/activate. On Windows it's a bit different, you call the activate script directly without using a dot, and it's in a directory called Scripts, not bin. After activation you can see the name of the active environment in the shell prompt and you can start working. Once you're done you deactivate the virtual environment with the deactivate command. Now we took a short look in the bin folder of the virtual environment and so we'll have the path changes, but that doesn't tell the whole story. There are some things I didn't explain because I think you should be able to figure them out by yourself. So please consider the following questions as exercises. I think you'll guess what the answer should be, but please start a command line yourself and check whether you're right. First of all, where do packages go when we install them inside a virtual environment, as opposed to system-wide? Secondly, what exactly happens to sys. path, the Python variable that controls where we look for packages, when a virtualenv is activated? Now if you watched the previous demo I think you know what to do to figure out the answer to these questions.

Virtualenv vs. Projects

Now I haven't really shown you a real project inside a virtualenv yet, but before I do a short word on that. The best practice is to keep your virtualenvs separate from your actual Python code. You might be tempted to put a virtualenv inside your project directory and then commit it to version control, but you really shouldn't do that. Basically your Python projects contain the actual source code you write, and that's the stuff you want to keep in version control. And the virtual environments contain stuff you need to work with your projects like the right libraries, and tools, and Python interpreter for each project. Now you keep all these virtual environments in a separate location like. virtualenvs in your home directory. Now as you can see in the picture, each of my projects in the dev folder has its own virtualenv in the virtualenvs folder. And of course, I always activate the right virtualenv before working on one of my projects.

Demo: Virtualenvwrapper

So let me show you my current development environment. As you can see, right now I'm in the dev folder, and that's where I keep all my projects. So that's my actual Python projects, and as you know my virtual environments are in my. virtualenvs folder. So those are separate. So I can look at all my projects like this, there's a whole set of them. And now to activate the environment from my amaze project, I have to call the virtualenv activate script with the dot command from. virtualenvs/amaze/bin/activate. And now the environment is activated and we can move into the project folder. But actually there's another package called virtualenvwrapper that makes life a lot easier for us. It gives me a command called workon. I can press step here and see a whole list of all my virtual environments. Actually there are some here that aren't coupled to projects, the p2 and p3 ones, and there's also some projects that don't have virtual environments because actually they're not Python projects. But okay, that's beside the point. Now let's say I want to switch to another project from my amaze, I want to work on my towerdefense game, all I have to do is this: workon towerdefense. And now the workon script actually does three things. First it deactivates the current active virtualenv, which is amaze, and then it activates the towerdefense virtual environment, but it also switches my working directory to my towerdefense projects. So right now, I'm in my dev/towerdefense directory. And now let me show you that the dependencies for these projects are actually completely isolated. So if I do a pip list here in my towerdefense projects we see a whole list here including let's say, Kivy, which gives me a nice user interface for games and pygame, and stuff like that. But I can also say, for example, I want to workon fabric. And again, that deactivates towerdefense, activates the fabric virtualenv, but it doesn't actually change my working directory, and we'll see why that is in a moment. And then if I do a pip list, I see a completely different list of installed packages. And that's because the virtualenvs are separate and all these packages are actually installed in completely separate locations. So for the towerdefense game all the dependencies are installed in my. virtualenv/towerdefense. And for fabric they're all installed in. virtualenv/fabric, etc, etc. And again, I can say I want to work on amaze, and I'm back again. And as always I can deactivate my current environment by saying deactivate.

Review: Virtualenvwrapper

So you just saw a more user-friendly wrapper around virtualenv and it's called virtualenvwrapper. It allows you to list all your current environments with the workon command if you don't give it any arguments. Or you can switch to a project and activate its environment with the same command when you provide workon with a project name. Now you can also create and remove virtual environments with the mkvirtualenv and rmvirtualenv commands. Now, of course, to use virtualenvwrapper you need to install it first. There's a couple of things you have to know about installing it. First of all, just like with virtualenv, you should install it system-wide. Secondly, the normal virtualenvwrapper script doesn't support Windows, so if you use Windows you should install virtualenvwrapper-win instead. The default location for your environments will be in a folder called Envs under your user's profile directory. Now, UNIX users can simply call pip install virtualenvwrapper but you should be aware that, again, you may need to use sudo. Then after installing, you need to add two lines to a file called. profile in your home directory, it looks like this. And I'll show you how to do this in a short demo. Now if you install virtualenvwrapper this way, the default location for your environment will be in a folder called. virtualenvs in your home directory like we already saw.

Demo: Setting up Virtualenvwrapper

So on Mac OS and Linux we install virtualenvwrapper like this: sudo pip install virtualenvwrapper. And on Windows you would do this: virtualenvwrapper-win and you would leave out the sudo part. So let's run this, I have to give my administrative password, very well, and like I told you we have to set up the. profile file, and before I do that I want to make note of where the virtualenvwrapper. sh script, which I have to read in, is located. And it tells me in the installation log, so I can copy the location here. Very well. So let's go to my home directory first and then open my. profile file. Now there may be some other code in here, as you can see I have a line that imports code from. bashrc. You may have different code here or no code at all, it doesn't really matter. What I do is first I say source, and I paste the location of usr/local/bin/virtualenvwrapper in here, that's what I just copied from the installation log. And then before I run the scripts I have to set an environment variable, and it looks like this: export PROJECT_HOME, $HOME, that means my home directory, make sure you use a dollar sign and home in uppercase. And in my case I say my project home is dev, that's where I keep all my projects. And of course, you can use any location you like actually here, as long of course, as it's where you keep your projects. So first we set a variable that will point virtualenvwrapper to where my projects are and then we import the virtualenvwrapper script. Now, and this will not be loaded immediately. One of the things you can do is restart your shell, so I would have to quit this window, and then restart shell. Or another thing that I can do is load my. profile like this, and now I can say for example workon amaze, and this will activate the virtualenv. But as you can see, right now it doesn't move to my project directory yet because I haven't bound the projects to the virtualenv yet. So let me explain a little bit more about that.

Demo: An Existing Project From Version Control

So basically when you're developing you normally have two things: a project and a virtual environment. And they're in separate locations and we're using virtualenvwrapper to manage that. So virtualenvwrapper knows where to look for projects and it knows where to look for virtual environments. The only thing that's missing right now is that we want virtualenv to know which project is coupled to which virtual environment. Now let me show you some typical usage. I want to work on the project requests, and for that I'm going to clone the requests library from GitHub. Very well. So now we have a new project here, we can move into it, etc, etc. And I can make a new virtualenv with the mkvirtualenv command. Now and as you can see, this is different from the virtualenv command we used previously because the virtualenv command would create a virtual environment in my current working directory. But the mkvirtualenv command from virtualenvwrapper will actually make it in my. virtualenv directory. So it will make it into the right location, I don't have to worry about that. And I can make it a new virtual environment called requests. And the nice thing is it doesn't just create it, it also activated immediately, as you can see at the start of my prompt. So that's all very nice, but let's say I deactivate this for now, and I move somewhere else, and I want to workon requests. Well that's nice. It activates requests, but it doesn't move me outside of my home directory and I'd like it to move into my development environment immediately. And we can fix this by moving into requests, like this, and then I say setvirtualenvprojects. And this couples my current working directory to my currently active environment. And as you can see it says, setting project for requests to /Users/reindert/dev/requests. So now it knows which project directory goes with this virtual environment. Again, now if I deactivate it and move somewhere else, what I can do now is say workon requests, and I'm not only in active environment right now, I'm also in my project directory, and that's what we want.

Demo: A Project From Scratch

So we just saw how you can start working on a project from version control, so that's an already existing project that you check out, and then you want to add a new virtual environment and bind that to the project. But sometimes you don't want that. Sometimes what you want is to create a whole new project from scratch. And there's a command for that as well, it's called mkproject. And let's make a project called, let's say, sample. And that does several things, it makes a new virtual environment, it makes a new project directory for you in your dev folder, in this case dev/sample, and it binds the two together, and it activates your virtual environment and moves into your project folder as well. So that does all the steps necessary to have a complete development environment ready for you.

Binding Virtualenvs to Projects

So we just saw that the commands setvirtualenvproject will bind an existing project like one that you just checked out from version control to a virtual environment. From that moment you can start working on the project anytime by using the workon command. There's also a nice utility called mkproject, and it will create an empty project, as well as virtualenv, and bind them to each other in a single step. And of course, it will immediately activate your virtual environment, and move you into your project's directory. Now for Windows users, as you know, you have to use a different package called virtualenvwrapper-win. And that has some small differences. The two main differences you need to know are, first of all there's no setvirtualenvproject command, but in this package it's called setprojectdir. But it basically does the same, it binds an existing project to a virtualenv. And another thing, and I think that's a pity, there's no mkproject command in virtualenvwrapper-win. So that means you can't create projects in virtualenvs in single step, you have to create them yourself and bind them with the setprojectdir command.

Resources

Now in case you want to do some more in-depth reading of the tools I showed you, here's the virtualenv documentation. Once you get used to working in virtual environments, you should really check this out to get a look at some of the advanced options. Then there's virtualenvwrapper. This too has some very nice extra utilities and commands that you should really check out once you get the hang of it. Now there's also a link for the Windows port of the virtualenvwrapper project, and there's some minor differences between this port and the main project. So if you are a Windows user, it's probably wise to check out the documentation for a moment.

Summary

And that's all I have to tell you about virtualenv. We've seen about the virtualenv package and how it's used to isolate your Python project so that installing packages doesn't cause conflicts. We've also seen how to create a virtualenv, activate it, use it, and then deactivate it. We've also seen the virtualenvwrapper utilities, which lets you switch between projects very fast, with the workon command, create new environments with mkvirtualenv, and bind them to a project with setvirtualenvproject. Finally, we saw that you can create a project and environment at once, with the mkproject commands.

Checking Your Code Quality With Pylint

Introduction

Hi, my name is Reindert-Jan Ekker and in this short module I want to show you a nice utility to help improve your code quality. You can make your life as a Python developer a lot easier if you know some tools that will help avoid some common pitfalls in your code, like when you accidentally mistype the name of a variable. And to make sure your code is consistent, and clear, and readable, you can also have it check whether you're following the standard Python code style. So in this module I will first introduce you to Python's standard code style as defined in a document called PEP8. And then we'll look into a code checking tool called Pylint.

PEP8: The Python Style Guide

The abbreviation PEP is short for Python Enhancement Proposal, and they are the place to look for authoritive of information about Python standards and best practices. You can find all kinds of information there like best practices, guidelines, and proposals for new features. The official description of a PEP is that it's a design document that provides information for a Python community, or describes a new Python feature. There are many PEP documents, but in this module we'll focus on PEP8, which is the standard style guide for Python code. Broadly speaking, it tells you how to layout your Python code, how to handle whitespace in your Python code, how to write comments, and how to name your classes, methods, variables, etc. Keeping to the standard Python coding style will make your code look clean and readable. In many cases I'd say that the quality of your code will improve once you start giving attention to your code style, because it will make you think about writing clean and attractive code. Now let's see some of the more important rules in this guideline. It doesn't make sense for me to cover everything here, but I do recommend that you take the time when you feel like it, and look through PEP8. No need to read it all, but it's a good thing to know what's in there, and maybe it's even more important to understand the underlying ideas.

Overview of PEP8

One of the things that PEP8 gives you is a convention for laying out your code. Now it's important to realize that these rules are about readability and clarity of your code, not about following rules. So if you disagree with some of the rules in PEP8, or for some reason you find that in some situation your code is clearer when you break the rules, feel free to do so. Now to give a high level overview of PEP8, I'm going to tell you some of the most important rules that are in there. To start, it specifies that we indent our Python code with 4 spaces per level, and that the preferred maximum line length is 79 characters. It also tells us to put two lines between top-level functions and classes, and one line between methods inside the same class. Then it gives some very detailed pointers about how you should import names from other modules and packages. I will not explain that in detail here, but you may want to read up on it sometime. Now, something else it talks about is how to use spaces in your Python expressions like when you say X equals 10. Again I think it's a good idea to read up on this. But in case you don't know the rules exactly, don't worry, most of the time our tools will tell us when we break them. Other important rules in PEP8 are about documentation and naming. The most important rule about documentation is that you should include docstring for all your public components, from modules, functions, to classes, and methods. I expect that this doesn't come as a surprise, there's a whole other PEP document, by the way, that describes what should go in docstring. So PEP8 basically only tells us that there should be documentation, not what should be in there. Then it has a lot to say about naming. Again there's a lot of detailed rules, but the main ones are modules should have lowercase names and they should be short, classes have capitalized names without underscores, functions and methods should have lowercase names with underscores between the words, constants should be in all caps, and any names not intended for public use should start with an underscore. Again, I expect that none of these rules are really new to you. The nice thing is, if they are new to you or when you make a mistake, there's a tool to check it. And then PEP8 goes into a list of general programming recommendations. This includes things like how to compare a value to none, or from which base class an exception should inherit, etc, etc. It contains a lot of little gems that are not just about style, but about writing code that is less error prone and more readable. And again I can't say this enough, I urge you to take a little time to look at PEP8. But for now I'm going to leave all the theory behind and take a look at how to apply this in practice.

Demo: Using Pylint to Check Your Code

So remember that some demos back I made an empty project called amaze. Actually that's a little pet project I've been working on, it's a procedural level generation library, but for now it doesn't do more than simply generate a little maze. And I'd like to show you this project as an example. Let's go and work on it. So the code in this project directory is a work in progress. I have a little demo script here, but if I try to run it, it gives an error. Now I've also already installed Pylint, as you can see. So let's see if Pylint can help us clean up our code and run our demo. Now I can run Pylint in two different ways. First of all I can run it as a command line script, like this, and in that case I have to specify the name of the module or package I want to check. So to check the code in all the Python files inside my amaze package, I say pylint amaze. And in that case, Pylint will go into the amaze package and check all the Python modules in there. So let me show you, here we are inside dev/amaze, and inside the dev/amaze folder there's the demo. py script and amaze package itself. If I open it, it contains some Python code. So if I call pylint amaze from this folder, it will go into this folder, find the package, and inspect all these Python modules. Now you should be aware that if you tell Pylint to check a package like this that your directory then needs to include a file called __init__. py, like this. Because if a directory doesn't have such a file, formally it's not a Python package, it'll just be a directory containing some Python modules. Now this file can be empty, or it can contain some documentation and other things, but it doesn't really matter for Pylint. So when I run this command, pylint amaze, it found this folder, and saw that it was a package, and then checked all the code in there. So as you can see it shows a lot of output and, to be honest, it's not really user friendly, but there's another way to call Pylint and that's with pylint-gui, which stands for graphical user interface. And it starts a very simple little user interface made in Tkinter, which is the default Python user interface kit, and unfortunately it doesn't look like much, but it works just fine. By the way, on Windows, this graphical user interface doesn't work when you install Pylint inside a virtual environment. So if you want to use this as a Windows user, try installing Pylint globally instead of inside a virtual environment. Now I can open a package, I can also open a single file, but for now let's open a package. And then I have to click Run. Now the first thing that attracts my attention here is the fact that Pylint gives my code a grade, a 7. 05 in this case, which is not too bad actually. Let's see what it warns us about and let's start with the most important messages. Apparently there's an error in my code, I made a typo in an assignment. I can double-click on this error and see where the error occurs. So let's open editor, and actually here on line 53 that's where the error occurs, and we see that we're trying to use the value of the _size attribute. But that does seem right and the actual error is here in the constructor where I made a typo, because the name of this attribute I'm assigning to should be _size. Very well. Then I can save it, and unfortunately for the graphical user interface, to pick up this edit I have to stop it and then restart it, and I can double-click on the history here and Run it again. And now, at least, the fatal message has disappeared. And you can also see that my grade is now raised from a 7. 05 to a 7. 77, so we're actually improving here. Now let's check some of the other errors, let's check some of the warnings. And apparently I import something I don't use. Let's remove that. It's in the kruskal file, let's open that, and let's remove this from the imports and let's run the user interface again. Now then there's the problem with the unused arguments warning. Well in fact, the function it complains about is this one, connected. In the maze module, let's look at it here, so the connected method here always returns true and it never uses any of its arguments. So Pylint is right that I have unused arguments here. But in this case the warning is unnecessary, this is exactly how I want the function to be. Of course, we can simply ignore the warning, but let's go one step further and configure Pylint to not show this one. To do so I can add a comment, like this, and let me just change the theme here to make it a little more readable for you. What I say is pylint: disable, and here we can tell Pylint not to show the unused argument message for this method. Now the nice thing is that Pylint will disable this only for the scope of the current block. So in this case it will only disable this message for this method. Now, unfortunately, the Pylint user interface does not tell us the name of the message, but if you run Pylint from the command line it will tell you. But this time let's tell it not to show all the information, but only the messages themselves, not all the statistics. I can use the -r switch which stands report, and then the n to say no reports, and then I tell it to look into amaze. And then I get not as many messages, which makes it a little more user friendly. And then here you can see the unused argument warnings have the name unused-argument. So if I Copy this and paste it into my code here, and then save, and then rerun Pylint, the unused-argument messages are gone. Now as I said, using a comment like this turns off the warning for the current block only. You can also turn off a warning for an entire file, I did so in the disjointsets file. At the top, here Pylint will complain about my use of short variable names like X and Y. In my opinion the use of these names is okay in this module, so I turn it off for the entire file. Now as you can see, Pylint shows a lot of messages and most of them make sense as well. Now we can make our score go up by either fixing the mistakes or telling Pylint to ignore them. Another nice thing is that Pylint detects code duplication as well, as you can see here. Now in case you have access to the course materials, I encourage you to download them and take a couple of minutes to see if you can get Pylint to give this code a 100% grade. Now finally let me show you that we can now run our demo without it giving an error message. So here we have a nicely generated maze by this little amaze package.

Pylint

So let's talk about Pylint. It's a program that does static code checking, which means it doesn't compile or run your code, but it simply looks at it and parses it. It does several things, mainly it looks for places in your code where you're violating the PEP8 guidelines and for so-called code smells, which means it looks at some patterns in your code that look like common errors. Another nice thing is that it detects code duplication, which means it will detect when the same code in your project shows up in multiple places. And basically, a lot of the time that means that some code has been copied and pasted from one class to the other. And usually that means that you can clean up your code by doing a little refactoring and make your code easier to maintain. Now in most practical projects Pylint will show a lot of messages and not all of them will apply to your code. Sometimes you have decided to deviate from PEP8 with a good reason, or maybe Pylint sees a code smell, but the code is actually correct. Fortunately, Pylint is very configurable and it offers several ways to suppress and filter messages so you'll only see the messages you want to see. By the way, there are many IDEs and code editors that offer integration with Pylint, so you won't even have to run it yourself. But for now I do want to show you how to run Pylint from the command line so you know at least how to interact directly with the tool itself.

Running Pylint

So the basic use of Pylint is to call it with the name of the module or package you want to check. As we saw in the demo, you can suppress a lot of the statistics by adding -r n to the command line. You can also use the --help option to see all available options. There's quite a few of them. Another very nice thing is that Pylint comes with a basic graphical user interface, which, I grant, isn't very pretty, but it does give you complete control over the information it shows you, and you can click on a message to show the actual Python code it's complaining about. You can disable messages by adding a comment in your code, like this, pylint: disable, followed by the name of the message. But sometimes you want to disable a warning globally and you can do that too. When you run the command pylint --generate-rcfile it will generate a configuration file with lots of options and you can globally disable messages, tell Pylint what your naming conventions are, configure reporting, and more. Now the example command here ends in a greater than sign followed by a name, pylintrc. This means we are saving the output from the command line to the file pylintrc. Now if Pylint finds a file like that in its current working directory, it will read it and apply your configuration. By the way, Pylint will not only search for a pylintrc in the current working directory, but also in your home directory in case you want to set global settings for all your projects, not just for one particular project, or per package so you can put a pylintrc in a Python package and specify pylint options for that package only. So let's put this into practice.

Demo: Pylintrc

So I can call pylint like this with --generate-rcfile, and it will output a default rcfile. It generates a lot of data and let's inspect what it generates. Well, let's start by giving some options for tweaking the way Pylint runs. And I'm not going to go into those right now, and some deprecated options as well. And then here we have something called MESSAGES CONTROL, and here you can enable and disable various messages in a global way. So for example, to globally disable the unused names warning, I would put that on this line so this line would then read disable equals unused warnings. Of course, I would have to uncomment it as well, so I would have to remove the hash sign here. Then I have a lot of options to tell Pylint about the kind of reports I want to see. So the -r n option I just showed you can be configured here by putting no into reports, and if you do that then by default Pylint will only show you its messages. So going down a bit, there's some very nice other options here as well. For example, here are some function names that should never be used because they're in the global namespace. And naturally, if there are other function names that you disapprove of, you can add them here. And in the same way, here are good names, variable names that should always be accepted. So in my case I could add X and Y here, and I wouldn't have the errors that I have now. Going even further down we find a group of regular expressions that match the PEP8 naming conventions. So here, for example, it says that function names contain letters or numbers separated by underscores, and the first character of the name should be either a letter or an underscore, so not a number. And you see the same thing here for variable names, constant names, etc, etc. So you can tweak all these settings so if you disagree with PEP8, or you use a different coding standard, you can configure Pylint to cater exactly to your needs. So as you can see there's lots of things that you can tweak in Pylint and as shown in sheets, if you want to do that you call pylint --generate-rcfile, and then use a > sign to save the output, pylintrc. And now you're ready to start editing the file with your favorite editor.

Resources

Now here's a couple of links in case you need some more information. Of course, here's a link to the PEP8 Style Guide document. And this link takes you to an overview of all available PEP documents. For more information about Pylint, visit www. pylint. org.

Summary

So in this short module I told you a bit about the PEP8 style guide and we looked into the Pylint tool to detect code smells and style violations. Now let's move on to the next module where we'll see a tool that will help you fix broken code.

The Python Debugger

Introduction

I'm Reindert-Jan Ekker and in this module we'll look at the standard tools available for debugging your Python code. Python comes with a module to help debug your code and it's called pdb, The Python Debugger. We'll examine how to use it interactively while your program runs so you can inspect the state of your program while you step through each line of code as it is executed. And you can even make changes and see what the effect will be. Now most of this module we'll spend on the command line again because that way you interact with the debugger directly. And using the debugger from a command line is a skill that's very helpful in cases where you don't have the luxury of using a graphical user interface. Like for example, when you have to work remotely over an SSH connection. But of course, there are many IDEs that provide a graphical user interface for Python debugging as well. And we'll have a short look at that too.

Demo: Starting a Debugging Session

So let's start with a simple debugging problem. I'm still working on my maze generator and I wrote a little demo user interface so I can look at the maze, and I wrote a maze runner as well, which will solve the maze for me. Now like with the Pylint user interface, I implemented my user interface with Tkinter because that comes with Python by default. Let's have a look. Now, again, it doesn't look too pretty but for a simple demo it'll do. I can generate a nice maze like this, let me select some nice dimensions, very well, and then when I click Draw it generates a maze. I can also select a different algorithm and we get a different kind of maze that looks kind of different. And I can also ask my algorithm to solve the maze with the Run Maze button, but that doesn't really run. Actually it enters an endless loop and I have to quit the program. So how do we fix this? Well let me know you the code for the maze runner first. So basically here's the run function, it create a Runner instance, and then calls step on the runner for as many times as necessary until we reach our destination. It also yields the runners location after every step. Now of course, I can try to find the bug by putting print statements in here and seeing what the program does, but that will mean a lot of restarting and rerunning the program, and changing print statements around. And actually there's a better solution. What I can do is insert two lines here, and the lines are import pdb, which will load a module, and pdb. set_trace, which will run the debugger. Or actually it will start an interactive debugger session. Now this is executed as soon as program execution reaches this point in the program, and I wanted to start after the runner has been instantiated and just before we start actually in the while loop to solve the maze. So, let's save this and run my demo again. Very well. I'm going to make a smaller maze so we can have a better look at what it does. And now when I click Run, here we see a debug message appearing. First I see that the debugger tells us that we're in the module runner. py, on line 33, in the function run, which is here. Then it lists the current line, which is of course the same as this line. No in case you don't have the actual file open while you're debugging, and that happens a lot, you can use l to list the current line of code with 10 lines of context around it. And as you can see the arrow here denotes the current line, and the rest is some context. Now in an interactive debugging session I can also run any Python code I like. And that Python code will be evaluated in the context of the running code, so it will run in the local context of this function. So for example, the local runner variable here is actually accessible to me just like local variable. So we can look at it and, for example, ask for its location. And of course right now it's at location 0, 0.

Demo: Simple Debugging Commands

Now to execute the current line, which is the start of a while statement that loops until we reach our destination, I type n, and that's short for next. It will run the current line of code and then move on to the next statement. So now it tells me we're on line 34 of run, and that's the line runner. step, as you can see. Now I can type n again, and it will execute the function called runner. step and move on to the next line. But actually I want to know what goes on inside the runner. step function. So I don't want to use the n command right now because that will step over the whole function call, I want to actually step into that function, so I don't just want to call it, I want to move down into the execution of that function. And for that there's the s command, which means step into. So if I do that we see that pdb tells us that we just did a function call and we're now in the step function, on line 56, of runner. py. Now of course, I can use the l command to look at the current function, or I can simply scroll down here, doesn't really matter, in both cases we see this function. Now the step function is the core of the maze runner functionality. Basically it runs the maze by always keeping its left hand to the wall, and that will guarantee that we solve the maze. So imagine you're walking through a maze in the dark with your left hand to the wall, if you take a step forward and then suddenly there's no wall at your left hand anymore, that means the wall has ended and there's a corner. So you turn left, and then you take a step forward, and you'll have a wall to your left hand again. Now if there is a wall to your left hand, then you'll move into the else branch here, where we check if we can actually move forward. And if you can't move forward that means there's a wall in front of you, and in that case we'll have to rotate to our right. And we'll do that until we can move forward again. So this simple algorithm is all it takes to solve the maze, and yet it doesn't work currently. So to solve why it doesn't work, let's start by inspecting the state of the program. As you know we can execute Python code here so, for example, I can ask the runner instance for its x coordinate, or I can use a tuple and ask for multiple values at the same time. Again the statements I run here are evaluated in the context of the running code, so right now self here points to my runner instance. So now we know that we're at point 0, 0 with direction 0. But the direction is actually an index into an array, and if we move up here I can show you, there's a DIRECTIONS array here, so if we have index 0 that means that currently we're pointing east. Now let's make our state here a little easier to read. Very well. So now we can see that we're at 0, 0 and our direction points east. So listing where we were again, we're at the start of the step function, and let's move into it with the next statement, and then we're here, if not self. _wall_at_left_hand. Of course, currently we're facing east and we're at the top level, so we're basically pointing right, and that means that there is a wall at our left hand. Let's execute this line and see whether the program agrees. And since apparently we now moved into the else statement, because our current line is now while not self. _can_go_forward, apparently the program has correctly determined that there is a wall at my left hand and we moved into the else branch. So let's look up the same code in the IDE here, there is a wall at our left hand, so now we're here, and looking at the maze we clearly can go forward as well, so I expect the program to do so. Let's see what happens. Yes, exactly, we move down onto the self. _move_forward line, so let's move forward, and now after calling move forward, pdb tells us that we're going to Return from this function. Now before we do that, let's check that our coordinates have actually been updated, that we have actually moved forward, and I can use the arrow up key to recall earlier commands. So I won't have to retype the whole commands to look at the state of our program, I can simply type up multiple times until I find the command that prints my coordinates. So we moved forward, it hasn't been drawn here yet, but we actually moved forward and our current location now is 1, 0, so that will be this spot. Now pdb also signaled us that the method is about to return because this was the last statement in the method. So what will happen when the method returns? Well, we'll see that in a moment, but first let's review what we just saw.

Review: Simple Debugging Commands

So you can add these two lines anywhere in your code to start an interactive debugging session: import pdb followed by pdb. set_trace. Now remember that set_trace is a function, so add parentheses to call it. We saw several commands for the debugger as well: l to list the current line with some of the code around it, and to execute the current line and move the program to the next statements type n, and one we didn't see, but is good to know anyway is h for help. Now I also showed you that you can use the up and down arrows to recall lines that you've typed into the debugger prompt earlier.

Demo: Advanced Debugging Commands

So we move to the right and now the method is about to return, this means we'll step out of the method and into the function that called it. We can ask the debugger to show us which method that will be. I can use the command where, or w, to print the call stack for the current program state. And we see that we're on line 63 of the step method, which was called by the run method in line 34, which in turn was called by run_maze in the tkdemo module on line 179, and that in turn was called by the Tkinter framework in its main loop when I pressed the Run Maze button. So now that means that when I say next the function will end and we will end up one level higher in the stack. So we will end up in the run method on line 34, let's see if that actually happens, and yes, of course, we ended up in the run method and not on line 34, but one line lower on line 35, which is at the top of this file here. We just called step and now we're going to yield the current location of the runner to our calling function, and we saw that that was the run_maze method in tkdemo. py. So now if I say n, for next, first pdb signals us that we're going to return from the current function. And then when I say next I end up in the run_maze method. Now again, of course, we can call l for list to look at the current code, or we can call w to look at the current stack trace. And of course, the current stack trace will be shorter because it won't contain the two methods that we just stepped out of. Now a nice thing is that the self. update_runner method will update our user interface, so now if I say n we see a new breadcrumb appear here in the maze for our new position. So all of that is very nice, but actually I'm more interested in fixing my algorithm. I can press c for continue and the debugger will let the program flow continue normally. Now I'd like to do that and start debugging again when execution enters the step function again. And to do that first I will set a breakpoint to start a new interactive session as soon as we reach a specific line of code. You can set a breakpoint with the b command, and let's ask for help about the b command with h, which stands for help. And as you can see we can set a breakpoint with a filename, and a line number, or a function. And let's start by using filename and a line number. I know that the step method is on line 58 of my program so I can say, this will set a breakpoint on line 58 of my runner module. Very well. So now if I press c for continue, the program flow will continue without debugging until we hit this line of code. Very well. So now I'm in my step method again, listing the code we can see that a breakpoint is set in this line now because there's a b here, so let's go on and debug. There is a wall on our left hand, we're still facing east so this will be false, and we will end up in the else statement. Let's check. I'm going to do n. Very well. So we're here now, while not self. _can_go_forward. And actually I'm facing a wall here so I can't go forward, so I expect to enter the while loop here. And that's exactly what happens. So the call to self. _rotate_right should rotate our direction 90 degrees to the right, and before calling it let's see what the current state of our program is. Very well. Currently we're facing east, now let's run the rotate_right method by saying next, and let's ask for our new state. And now actually we're facing west instead of south, which I expected because of course if you're facing east and you rotate for 90 degrees you would face south after, but actually we're facing west here. So apparently the bug is in the rotate right direction because it didn't rotate 90 degrees, but it rotated 180 degrees. So let's set another breakpoint in rotate_right this time. This time instead of using a line number I'll use the method name. So the breakpoint is set and now I can continue to see the runner walk until it has to turn right again. I'm going to say continue, and we end up in the next call to our step method, and actually now it faces left again so first we'll step back. Very well. And now you can see it move down, moving down again, moving down again, it has to turn left here. Very well. And now we're facing a wall again and we have to rotate right. So now if I say continue, we end up in our rotate_right method. Now of course, if we examine the stack trace we see that we're in the rotate_right method, which was called by step, which was called by run, etc, etc. Now, this is the actual line that contains our bug. It takes a current direction index and adds 2 before taking a module of 4. Now executing this, and looking at the new state, we see again that it turns around 180 degrees. And actually, what I should do, is increase the direction one more, and I can do that right now. Very well. And now, actually, we're facing south. So that's our bug, instead of adding 2 here, I should add 3, and that will make us rotate right. So let me fix that here. Very well. And I'm going to quit the debugger here, and then of course, don't forget to remove our debugging lines here, and let's run the demo again. And now you see the maze runner runs and solves the maze.

Review: Advanced Debugging Commands

So some other commands you can use in an interactive debugging session: first of all there's c, which will continue normal program flow without debugging until you hit either a breakpoint, or a new set trace call. B to set a breakpoint and you can set breakpoints at lines, or at specific functions. W, which means where, which will show you all the function calls in the current program stack trace. S, which means step, which will step into a function and take you down a level in the stack trace. And then there's one we didn't see, but it's r and it means return, and it will make you step out of this level. So when you're examining a function and you want to run to the end of the function, and then move up to the stack level above, use r for return.

Demo: Using the Debugger From an IDE

Now of course, most of the time you'll probably be using an IDE, an Integrated Development Environment, and that will probably give you access to a user interface for the debugger. For example, here I'm using PyCharm and here's the debug view. And in most IDEs you can click in the margin to set a breakpoint, so as you can see I can toggle breakpoints here. And then to run the program I have to tell the IDE to run our program in debug mode. And in this case I can do that by right-clicking on demo. py, and say Debug demo. And now it says my process is running, I can again Draw Maze and Run it, and as you can see we entered the debugger here. And one of the things it does is it lists all the local variables here, so here's my runner instance, and here I can see the current location, and the direction. So actually graphical user interface makes it a lot easier to use the debugger. Now here's something nice that you can't do with the command line debugging session, and that's something called Watches. I put two watches in here myself, there's self. location and self. _dir. And in the current context they don't exist, but if I put a breakpoint in the step method, here, and then I tell the debugger to continue running, we end up here in this breakpoint, and now these watches have values. So now I can see the location and the direction immediately. So with watches you can automatically keep updated on values you find interesting in your debugging session. And of course I can tell the debugger to go to the next statement, in this case the command is called Step Over, and of course, now if I rotate right I see the direction changing. And just like the command line debugger, there's lots of options like Step Into, or Step Out, which we called return previously, and that way you can basically do everything we just did on the command line with a graphical user interface. Now of course the exact functionality of your IDE and how it interacts with debugger depends on your development environment and I can't really go into detail here.

Resources

Now this was a rather short introduction into the Python debugger, and there's actually some more advanced things you can do with it. And if you want to know more, first of all of course, here's the official Python debugging documentation. But I also selected a very nice article, it's from 2005, but the Python debugger hasn't changed that much, so it's still up to date.

Summary

And that brings us to the end of this module. Basically what we saw is how to use pdb in interactive use when you use pdb. set_trace. And we also took a short look at using pdb from an IDE.

Documenting Your Code With Sphinx

Overview

Hi, I'm Reindert-Jan Ekker and in this module we'll see how to generate beautiful code documentation with a tool called Sphinx. So let's say your first project is nearing completion, and it's a library that you mean to be used by others in their projects. Now for them to be able to easily use it you have to have readily accessible documentation that's easy to read and to search. To make the task of documenting your code easier I'll show you the toolkit used by most large Python projects to maintain their documentation and to generate documents in multiple formats like PDF and HTML. We'll start by taking a short look at Python docstrings and the standard conventions for writing them. Then I'll introduce you to Sphinx, a tool that can generate nice HTML documentation. Sphinx is used to generate the documentation for the Python language in standard library, and it's a de-facto standard tool in the Python world. Among other things, it will transform the docstrings in your code to beautiful HTML. Now Sphinx also uses a markup format called reStructuredText for laying out text, and we'll look at that as well.

Docstrings and PEP257

There's a PEP document with number 257 about Python documentation. It tells us about the conventions and semantics associated with Python docstrings. In other words, it gives pointers about how to write clear and useful documentation. Like with PEP8 these are no hard rules, and as the PEP itself says, if you violate these conventions, the worst you'll get is some dirty looks. Now in this module we're concerned mainly with the tools for exporting the documentation to HTML, not with writing good documentation. So I'll not tell you about all the details in this PEP document, I just think it's good for you to know that it exists and let's only take a short look at the most important rules concerning the form of your docstring. I won't be going into the contents of the docstrings at all, because that would be something for a different course. Now, docstring, as you know, is a string that stands by itself as the first statement of a module, function, class, or method. The interpreter will read it and set it as the __doc__special attribute for that object, making it accessible within a Python program. The most important rules for docstrings are simple. You should always surround the docstring with triple double quotes to make sure all characters in the docstring are escaped, and you're docstrings should contain grammatically correct sentences.

So a short docstring should at least be a single phrase ending in a period. Now for methods, make sure that you specify the return value of your method in your docstring because that can't be determined by introspection at runtime. We can ask a method at runtime about the number of arguments it takes, for example, but we can't ask it for the kind of return value it returns. So that's why you should add that to your documentation. So what does a docstring really look like? Let me show you the 2 simple examples from PEP 257. This is what a single line docstring looks like. Notice how it's all on a single line and the use of triple quotes. Also notice how the documentation is a complete grammatically correct sentence ending in a dot. Now this is a longer docstring, using the three quotes will make new lines part of the docstring so you can safely write multiple lines of documentation. After the first line the text is indented to the same level as the quotes. Tools that process Python documentation will remove this indentation so that all lines will get the same indentation as the first line.

Sphinx

So that's docstrings for you. They're just an addition to the Python syntax to make it easy to document your code. Now to move from writing docstrings to creating beautiful, and readable, and publishable documentation, we're going to use a tool called Sphinx, which calls itself a Python documentation generator. But it's not just some projects, it's actually the de-facto standard for generating your Python documentation, and it's used by many large projects, but most importantly, it's used by the Python project itself to generate the official Python documentation. As its input format, it accepts something called reStructuredText, which we'll learn more about in a moment. From this input, Sphinx can generate all kinds of output, like HTML, PDF, eBooks, UNIX man pages, and more. Sphinx is a very full-featured tool and if it doesn't have the features you need, chances are that there's an extension that provides it. Now all of that sounds very nice, but the best way to show you the power of Sphinx is by running it. So let's see a little demo.

Demo: Sphinx Install and Setup

So, as always, we start by using pip to install Sphinx. Of course, I'll first activate my virtual environment, and then I call pip to install Sphinx. So one of the things this does is it installs a script called sphinx-quickstart. Now before running it, let's make a new directory where our documentation will go, and I'm calling it docs. Let's move in there, and now we can run the quickstart scripts. This script will create a directory structure and some files for us, and it asks us some questions about the configuration. Most of the defaults are fine and I can just press Enter. Now we need to enter the name for our project and the author name, as well as the current project version. And from here we can press Enter again several times, and now it starts asking us about extensions to Sphinx that we can install. I'm not going to activate most of them, but of course you can if you like. This, for example, is an extension to generate epub files, and I don't need it at the moment. But it is very important that you say yes to this question because you really want to use autodoc. This is the extension that will allow us to use the docstrings from our code to generate HTML documentation. But from here on I'll just press Enter for every other question. So let's see what Sphinx created for us. First of all there are three directories here called _build, _static, and _templates. These are used by Sphinx when it generates output. The _build folder is where the end result will be. Then it contains a file called conf. py, this is where the Sphinx configuration goes. You're free to edit it yourself and actually we'll have to do exactly that later in this module. There's also two files called Makefile and make. bat. We'll use them to run Sphinx. Makefile is for UNIX-like environments that come with a tool called make, like Linux and Mac OS. We'll see how to use those in a moment. Then there's index. rst, this is a restructured text file, and it's where we'll start writing our documentation. Let's open it. Now this is a simple text file in a format called reStructuredText. Although it's a plain text format, Sphinx uses it as input to generate HTML, PDF, and more. For now, let's just add a little text here below the main title. This text I just added contains three subheaders, Simple use here, How it works, and Features. Some code examples, like this here, and this here, and several paragraphs. Let's see what Sphinx makes of this. On the command line of course I have to make sure that I am in the docs directory, and then I can say make html, which on UNIX system will call a standard tool called make, which will run a code from the make file, and on Windows it will run the batch script called make. bat. This runs Sphinx and now Sphinx tells me that the HTML pages are in _build/html. So let's open that. Very well. I'm going to double-click index. html and this is the output. It contains a Table of Contents, a search box, and the text I wrote. So here is the Simple use subheader, here's some of my example code, and here are the other subsections. Now the current HTML theme may look familiar to you because it's used by the Python 2 documentation as well. Just for fun, let's open conf. py for a moment. As you may remember, the conf. py file is generated by the quickstart scripts, and if we scroll down here we find a setting called html_theme. Now there's another theme I like better than default and it's called haiku. Actually there's many more and you can find them on the Sphinx homepage. Now if I run Sphinx again, I can refresh my browser, and we have the same documentation in a different theme.

A Note for Mac OS Users About Make

Now just a quick note for Mac OS users, the make tool may not actually be installed on your system by default. Now there's a simple way to install it and you can do that by running this command, xcode-select --install. If you run this it will install a set of command line tools for developers. Pressing Enter, I get this pop-up and I can press Install, of course I have to Agree, and in my case I get this message that it can't install the software because it's not currently available. Well, actually, although it's a strange message, this message is okay because it means on my system that the tools are already installed. If you don't have these tools installed yet you can expect the installer to run correctly, and afterwards you can use the make command line tool.

Review: Running Sphinx

So after installing Sphinx we can run the sphinx-quickstart command to generate a base Sphinx configuration. The usual practice is to do this in a new folder called docs. The quickstart script will ask you a lot of questions, and for most you can go with the defaults. But don't forget to enable autodocs because that extension will let Sphinx import documentation from the docstrings in your code. Now after the script is run you can edit the reStructuredText and then run make html, which will generate the html output for you. By the way, the configuration choices you make when the quickstart script runs are stored in a file called conf. py, and you can always edit that afterwards if you need to.

Demo: reStructuredText

Now let's take a closer look at the reStructuredText. It's a plain text format that uses markup in a very natural way. The first rule you have to know is that blank lines are very important in reStructuredText. As you can see, all paragraphs have to be separated by blank lines. Code examples, like here, are surrounded by blank lines as well. Now you can add sections by underlining the headers. ReStructuredText is quite smart about figuring out what the first and second level headers are. Basically all you have to do is underline them. You can use almost any convention you like. I chose to use equal signs here to mark the top level header, and minus signs, like here, but I could just as well have used another convention. So for example, if I change it like this and use tildes for the main header with an over line as well, and I use plus signs for subsections, that will generate exactly the same output. Now for a code example you end the current paragraph with two colons, like here. Now make sure that you surround the code example with blank lines, so there's a blank line before and a blank line after, and to indent it as well. If you forget one of the blank lines or the indentation you'll get unwanted results and basically reStructuredText won't recognize your code as a code example. Now if you do it right, like this, and you look in the browser, you will see that the example is nicely marked and that Sphinx added Python syntax highlighting. Now here and there in my document I added emphasis to some text, like here the Python Developer's Toolkit, by surrounding some text with stars, and that will make the text in HTML show up in italics. Now to make the text bold I can add another star like this. And of course, I have to do that before and after the part I want to make bold. And as you can see my editor shows me immediately that that's the way it will show up. You can also add lists in many ways. Scrolling down, here you see an example of a bulleted list. In this case I used a minus to tell reStructuredText that it's supposed to be a bulleted list, but you can also use plus signs or stars if you like instead of the minus signs. Now the last basic reStructuredText feature I want to show you for now is that you can add hyperlinks very simply. Going up again, the link to the Pluralsight homepage is between back quotes, with the actual URL within angled brackets, here, and the part before it is the descriptive text shown as the link. So in the browser we will see Pluralsight and it will link to this URL. Now finally, notice that the first paragraph of this document was generated by Sphinx, and it starts with two dots. The four lines are grouped by indenting them so reStructuredText will know they belong together. Now the two dots like this mean that the paragraph is a comment and it will not show up in the output.

Review: reStructuredText

So reStructuredText is a plain-text based markup language. Two important basic rules are that paragraphs are separated by a single line and that indentation matters. Most of the time you will use indentation to group stuff together. You can emphasize a part of your text by using stars, single stars meaning italic text, and double stars meaning bold text. And one thing we didn't see in the demo, you can show inline code by using double backticks, like shown here. We also saw that you can use links like this, with back quotes, and the first example shows you like we saw in the demo, a link text where the text to show differs from the URL, and in that case you put the URL in angle brackets, and the link text before it, and don't forget to use an underscore at the end of the link. Now in case you have a simpler scenario where you want to show the URL you're linking to as it is, in your output you can simply put the URL in your text and it will be recognized as well. So in that case a link will be made automatically and you don't have to use any extra syntax. ReStructuredText lets you choose your own convention for breaking up your text in sections. So as an example here, you can choose to use equal signs for your main section title, and as you can see you can use over- and underline, but you could also choose to only underline, and that will work as well. And here I chose to use a minus for A section, and tildes for A subsection. You can basically choose any convention you like because reStructuredText will be quite smart and most of the time it will do exactly what you mean to do. We also saw how you can use bulleted list items with stars, or minus, or plus signs.

Demo: References

So let's take it to the next level and see how to make structured documentation with different parts that are interconnected by links. Now, we're starting again in the index. rst file and I added a single line here which says, For more information about how to use this library, see the API. And there's a special syntax here, and basically this is a link syntax which is very similar to the link syntax we saw for a URL link for the world wide web. But in this case it's kind of an internal link which will link to a different part of your documentation. And this ref here is a keyword that says we're adding a reference, and here, API, is the actual name of the target we want to link to. And I added new file called api. rst and this file will be picked up automatically by Sphinx, but it's not enough for the link to work. If we switch to the file, in the first line here you see I added some code, and this is actually what we call a link target, and that's how Sphinx knows where to link to if we say we want to link to the API. So if we start a line with two dots like this, that means we're giving Sphinx a directive, and in this case it's a link target. Now link target is given by an underscore, and then the name of the target, which in this case is API, and then a colon. So basically when in index. rst I say: ref:'api' like this. This will generate a hyperlink to the target defined in the other file with this directive. So the fact that my file is called api. rst doesn't really have an effect on how my links work. Now the rest of my API file is structured like the index file, paragraphs, headers for sections, we have a main header here which says API Documentation, I have subheaders like Generating a maze, I actually have sub-subheaders for which I chose to use tildes. So basically here we have three levels of headers. Now I've added sections for the various classes, functions, and constants I want to document. So if I scroll down a bit we see here that this section will document one class, and this section will document the Recursive class, and this will document the Maze class, and going down this section will hold documentation for the run function. Now I didn't include any actual documentation for these classes and functions here, but in a moment we'll see how to fill that in automatically from the docstrings in our code. But for now I want to focus on using references for a moment. So I have marked all my sections with a directive that tells Sphinx what kind of structure it's documenting. Now marking your sections like this will do several things. First of all it allows you to link to that specific section in a semantic way. So as you see here in the first paragraph, I can now link to the documentation for the generate function here with a special syntax. I use the func keyword, which says I'm linking to a function, and the function I'm linking to has a name, generate. And this will generate a link to this section here, which is marked with this function directive. And again, the link syntax is familiar. I have the back text here that gives the name of the target I'm going to link to, and I use a word between colons that tells me what kind of structure I'm referencing. Now similarly we have links to classes, and we can have links to other Python constructs as well. Now finally, here's a plain link made by simply putting back quotes around a word, and adding an underscore. It will link to a target called constants, and as you see if we scroll down, here at the end of the file, here's a link target called constants. And this is exactly the same syntax we used at the top of the file for making a link target called API. But the way we reference it, if I go up again, is a little bit different. When linking to the API in index. rst, I used syntax with the word ref, but in the case of linking to constants I'm using just back quotes and an underscore. Now this works pretty much the same, but the links will look differently in the HTML output. Let's check it out. And here we're back on the index page, and scrolling down a bit, here's my new line in the index page. And as you can see my link to the API documentation doesn't simply say API, but actually as the link text it uses the title of the section I'm linking to. So if I click this link, we see that the title of this section is API Documentation, and that's exactly what we saw in the link text. Whereas for the constants link, I simply see constants. We can also see here links, for example, to functions. So this will link to my generate function, if I click it we move to the section for the generate function. Now going back to our index page again, there's another thing I want to show. First of all, there's an Index, we can click it, and it contains all the objects we're documenting. So I can click, for example, Kruskal here, and we will jump straight to the documentation for that object. And again, going back to the index page, we also see a table of contents here. Now going back to my reStructuredText, if we look in the index. rst, and I scroll down, here we see a toctree directive. This was generated by the quickstart script so I didn't add it myself, but what I did add myself, here, is API. And basically that's the name of a file I want to add to the table of contents. So here I say I want to add the API file to the table of contents. That also say let's use depth of two, and then if we go back to our HTML output you can see what that means. It's because we see here the first level, which is the top level title of my document, as well as the four subsections Generating a maze, the Maze class, etc, etc. But this third level, remember we had a third subsection level, isn't represented here. And that's because we're telling the toctree directive that we only want to have two levels in our table of contents.

Review: References

So one thing we saw is that you can generate links between parts of your documentation using the reStructuredText reference syntax. You mark a section as link target with this directive: starting with two dots, followed by a space, an underscore, and then the name of your target. After that comes a colon. This is the simplest form of link targets. You can then add a link to the target with the ref keyword between colons. After that you put back quotes with the name of the target between them. This syntax will put the title of the section you're linking to as the link description. Or even simpler, you can use an underscore with the name of the target between back quotes. In this case the link description will simply be the target name. We also saw the first steps to documenting our code. You can add literal code blocks by ending a paragraph with two colons. The next paragraph will then be laid out as code, including syntax highlighting. For this to work though you need to indent the code block and surround it with blank lines. There's also some special directives to mark your sections as documentation for Python structures. This will make sure you can link to those sections in a semantic way, as well as adding a nicely laid out header and adding them to the index. Now here's a table with the constructions you can use. For the targets you use two dots followed by the word function, method, class, or module, and then two colons, followed by the name of the object you're documenting. Now to add a link to such a section, you use two colons with a word between them followed by the target name between back quotes.

Demo: Autodoc

So let's take the next step and have Sphinx pull in docstrings from our code and add them to our HTML output. Now here's our api. rst file again, and let me show you what I changed, it isn't much. What I changed is that I took the function directive here and changed it into autofunction. And I changed class here, into autoclass. And here as well, and here's another autoclass, and there I added a members option. And here again, I changed function to autofunction. And there were remarks here everywhere that I had to add some documentation, but I removed those because the autodoc extension from Sphinx will use these autoclass and autofunction directives to pull the docstrings from our code and put them in here. Now before I show you what this looks like, let me show you one thing, and that is we can intermix autodoc documentation, here we will have the docstring for the generate function and we can intermix that with regular reStructuredText. So after we have the docstring here, I added a little paragraph of my own because I thought that would help the reader a little bit with some extra context. By the way, I could've added an automodule here as well, to add the docstring from the amaze module, but I chose not to because I think this text I'm using here is actually more clear than the docstring from the module.

Very well. So let's run make now to generate our documentation. But now looking at the output there's an import error here, No module named amaze, and that's because the autodocs extension to Sphinx actually has to import the code we are documenting to be able to extract the docstrings. And for that it needs to know where it can find our modules. And we can tell it by changing the configuration file. Here at the top of the configuration file, there's a comment here that tells us that if needed, for example for autodoc, we can change the path. So I'm going to comment out this line, and I'm not going to add the current directory because that's our docs directory, and there's no code in there, but I'm going to add the parent directory, like this. Saving and returning to the command line, I can say make html again, and now let's look at the documentation. And this is what it looks like now. The start is the same as it was, but if we look a little bit down, here we see stuff that actually comes from our docstrings. So this here is the docstring for the generate function. And moving down, this here is the docstring for one of our classes. And as you can see I added some markup to our docstrings as well. So moving back to the editor, for example, if we look in the generate. py file, here we see the docstring for the generate methods. And because it's a simple string there's no reason we couldn't add reStructuredText structures here. For example, there's a param keyword that we can use to layout parameters. So here I'm telling reStructuredText we have a width parameter and a height parameter, and a generator class parameter. And I can even use hyperlinks here to link to other structures from my code. So now that we've seen this docstring, going back to the browser, we can see where these links come from, because they're actually reStructuredText that I'm using in my docstring. Finally the last thing I want to show you, if I go back to my API file here, and scrolling down a bit to the Maze class here, I have added the members option, and that tells that autodocs extension through Sphinx, that it should add all the public members from the Maze class that have docstrings to our output. So again, going to the browser and scrolling down to our Maze class, here is our Maze class section, and here's the autogenerated documentation, and you can see that all the docstrings for the public modules are added as well. So now you basically have all the information you need to start documenting your projects.

Review: Autodoc

So I just showed you how you can use the autodoc extension to extract the docstrings from your Python code and add them to your Sphinx output. Now the first directive we saw is autofunction, and that will extract a docstring for a function. There's also an automethod, which will do the same, but on methods, which are basically functions that belong to a class. Then there's of course, the autoclass directive, which works basically the same, but the main difference is that it has a members option, which will automatically look up all the public members which have docstrings, and add those docstrings to your documentation as well. Now you can also add a list of members to the members option, as you can see in the second example, where I tell it to only include the member's width and height. And then there's another autodoc directive, and it's automodule. We didn't see that in action, but it basically works the same as the autoclass directive. It simply includes docstring from your module. Now like a class, the module has members, so like the autoclass directive, the automodule directive has a member option. Now to use all of this, don't forget to activate the autodoc extension, and you can do that when you're running the quickstart script. Now also don't forget that autodoc needs to import your modules, and that means you will have to fix the path at conf. py or otherwise you will get import errors when you run Sphinx.

Resources

Now like I said at the start of this module, Sphinx is a very full-featured project and you can do lots of things that I didn't cover in this module, for example, using images and tables. So if you want to know more, this is the place to go, the Sphinx Documentation at sphinx-doc. org. It may also be a good idea to check out the reStructuredText Homepage. And then there's a very nice reStructuredText Primer, which will tell you the basics about the format. And then finally, here's the documentation for the Autodoc Extension that will let you automatically generate documentation from your Python docstrings.

Summary

And that's it for this module. We've seen some basic information about docstrings. We've seen how to use the Sphinx tool, and how to install it and run it to generate HTML docs. And we've also seen the autodoc extension that knows how to extract docstrings. And to use all of that we also need to know about the reStructuredText format that allows us to layout rich texts. Now so far so good, we're almost done with the course. Let's go on and see how to package and ship our project.

Packaging and Distributing Your Project

Overview

Hi, my name is Reindert-Jan Ekker, and welcome to this last module of this course in which you'll see how to package and distribute your projects. So in this course we've seen how to install the packages your project depends on, how to set up your virtual development environment, how to check your code quality, debug, and document your code. And hopefully at some point, your project will be ready for shipping. So let me tell you about some of the ways you might do that. First we'll see that it's really easy to package your project with a toolset called setuptools. Then after packaging you can share your project with the world by uploading it to the PiPy package repository. Finally I'll give you a brief overview of what the possibilities are when you want to distribute your project as a standalone executable.

Demo: Preparing Your Project for Packaging

So let's make our project ready for packaging. It's really very simple; all I have to do is add a file called setup. py to my projects top level directory. And that file setup. py should start by importing a function called setup from the package called setuptools. Now this setuptools package will normally be installed by default when you use virtualenv and pip. Note that it's not a standard library. There's also a standard library that comes with the setup function, and that's called distutils. But actually, that's in the process of being deprecated, and the Python community is moving in the direction of setuptools. So instead of using distutils here, which you might still find in some examples online, you really should use setuptools. So make sure you say from setuptools import setup. Now setup is simply a function, and after importing it, what we do is we call it, like this. And then you can pass it a number of keyword arguments. In this example I only use the very basic keyword arguments that you really need to always include. So that means we pass it the name of our project, the current version, a short description string, the name of the author and my email, as well as the packages inside my project that I actually want to distribute. And in this case I'm listing two packages, namely the top level amaze package, which is in this directory here, and also the amaze. demo package that's inside there, which is a sublevel directory of the amaze package. And you should note that although it's a sublevel directory of the amaze directory, we still have to mention it here in the packages list, otherwise it won't get included. And that's basically it. All I have to do is import setup and then call it with some data about my projects, and then I can start packaging my projects.

Demo: Creating a Source Distribution

Let's switch to the command line, and of course, as always, I'm inside my active virtual environment, and now this is the command to package my projects. I call the Python interpreter with, as its first argument, the setup. py script. And then I tell it to make a source distribution. That means we're going to make a basic package that's really just an archive file that contains my Python source file. It also means I don't have any C code, for c extensions, that have to be compiled, because in that case I probably would want to use a binary distribution package. But for now we don't need it, and let's stick to the simple source list. Pressing Enter, the command here shows all the files it packaged and put into an archive. So let's take a look at what this created, and now you'll find there's two new directories in our project. First of all, there's the amaze. egg-info directory. This is used by setuptools when it generates your package, and it's safe to delete, so I'm going to remove that right now. Then there's a new dist folder, and that actually contains our new distribution. This is the actual package, and as you can see it's a tar. gz file which is a UNIX archive format, not completely unlike a zip file. I can double-click it on a Mac, it will unzip it, and you can see that our package actually includes our amaze package with all the Python code, as well as the egg-info folder, which will be used at install time with some information about the package and the setup. py script with a config file. So that's basically what goes inside our package when we create it. Now let me remove this again so we're just left with our package, and now you're basically ready to share this file with the world. So let's say you share it with someone else who has his own project with his own virtual environment, so I'm going to set up a temporary environment for a moment, and I'm going to pretend I'm someone else who just received this package, let's say, from an email. And after I save to the file system, I can simply install it with pip. So I can say pip install and then point it to the file I want to install. So this is basically a pip feature we haven't seen before, you can say pip install and then simply point it to a file. And then when I press Enter, pip installs my package. Very well. So basically all I have to do to make my project ready for packaging, is add this setup. py script and then I can run the setup sdist command.

Review: Creating a Source Distribution

So we've just seen what you have to do when you want to prepare your project for packaging. Basically all you have to do is add a file called setup. py to your project's top level folder. Now setup. py is simply a Python script and you import the setuptools library, and then call the setup function with a set of keyword arguments. And the absolute minimum is that you provide at least the name and version of keyword arguments. Now take care that you import setuptools and not distutils, even though some of the tutorials on the internet will still tell you to use distutils. But now days, setuptools is preferred. So let's look at another short demo to see some of the more advanced features of the setup scripts.

Demo: Advanced Features

So for a moment let's investigate a real world, more complex setup, from a real project. Here I opened the setup. py file for the Flask microframework for developing websites. And as you can see there's a whole bunch of settings here, a lot more than I showed you a moment ago, and let's go over them. There's the name and the version of the projects, there's a url that points to the GitHub repository where people can download it, there's the license, which in this case is the BSD open source license, the author name and email, and the description string. And then there's a long_description, and actually that's important to use here, because that will show up on the PiPy package index when you publish your package there. And in this case the author made the choice to include the docstring from this module, so if we scroll up for a moment, we see that the file starts with docstring, which is actually valid reStructuredText. And if we open the Flask page of the Python Package Index, we see that the description here is actually this docstring. Now of course, going back down, this is simply Python code so you could do anything you want here. You could, for example, also call a function here that fills the long description from a file, or generate text in a different way. But in this case the author has chosen to use the docstring and I actually kind of like that. Just like with my project, you can see we're including the top level package, and two sub packages, and then here there's something called include_package_data. And that means that the author wants to include stuff with the package that's not actual code, but that's data. Now when the setup script sees this setting it looks for a file called MANIFEST. in, and that's here. And if I open it you can see that it contains a couple of directives to tell setup. py which files should go into the package. So for Flask it tells setup to include the Makefile, and the changes in license file, and more. And the artwork directory and the tests directory, example and docs, and then there's a whole lot that it excludes. For example, the pycache files from the docs and the test directories don't have to be included in the package. So basically if you want to include other things than just your Python code in the package, use the manifest. in file and set include_package_data to true. Then the next setting I want to focus on is the install_requires here. And this is actually a list of the requirements for installing this package. So when we will run pip install later, pip will see that we need the Werkzeug and the Jinja packages as well. So it will automatically download and install those. So basically these are install time requirements. Now don't confuse this with the requirements. txt file that we talked about in one of the previous modules because the requirements. txt file is about development time requirements. So that's what a developer might want to install when he wants to write code for your projects. But this is installation requirements and those are used by pip when you use pip install to install this project. And it will automatically download and install these other packages. Then there's a list of classifiers, and you can think of this as a list of tags that provide general information about the projects. For example, it tells you what the current Development Status is, and apparently it's still in Beta. It also tells us that this is a web development project, and that the Intended Audience is Developers, etc, etc. And one of the uses of these classifiers is that you can browse the Python Package Index by these classifiers. So they basically provide context information about the projects. So that's a little more complicated, real world example. Now there's another example I want to show you and that's a sample project provided by the Python Packaging Authority.

Demo: The PyPA Sample Project

The Python Packaging Authority have created a sample project that basically shows you a skeleton for a Python project, and I really like this because it gives you a very good starting point for setting up your own projects. One of the things in here, and probably the most important things is the setup. py file. And if you look at it, I'm going to zoom in a bit, you can see that all the important settings are in here and they added comments as well, so they actually tell you want all these settings are for. So instead of having to search in documentation, they provide you with lists of possible options and what they do. So that's a very nice starting point for your own project. Going back we see that they also provide a MANIFEST. in file, a README reStructuredText file, and a setup configuration. So like I said, this is a very nice sample project and I would advise you to download it and take a good look at it, and maybe even use it as the starting point for your own projects. Now of course you can look at the URL here, but I'll also include it in the resources slide at the end of this module.

Review: More Setup Features

So we just saw a whole number of options you can give the setup function, but maybe the most important one is the install_requires setting because that tells pip which other packages your package depends on. And if you forget it, people won't be able to install your package. So that's why I think it deserves a special mention here. Then we also saw that you can use a file called MANIFEST. in to explicitly include or exclude other files than just your Python code from your package.

Demo: Distribution With the Python Package Index

Now we've already seen how we can prepare your project for packaging with the setup. py file, and we've seen how to package the project with the sdist command. Now of course, you want to share your project with the world, and of course you can email your package to other people, but you probably prefer to share your package with the world by uploading it to PiPy. And we can do that very simply form the command line. We only need two commands. The first thing we have to do is say register. Now the first time you run this command, this is what you see, we need to know who you are. And in this case, probably you want to say two, you want to register as a new user, and you can create new login credentials, and you'll get an activation email. Of course, I already did this and I'm going to use my existing log in. Another setup script registers my amaze project to the Python Package Index. And that means that now if I search on the Python Package Index for something called amaze, we find my amaze generation library. But when I click on it, there's no downloads here, and that's because I didn't upload my project yet. So going back to the command line, let's try the upload command for the setup scripts. But the upload command doesn't work on its own, you have to tell it which kind of package you want to upload. And in my case that's a source distribution. Now after fixing my command in this way I can press Enter, and my package has been submitted to the Python Package Index. So refreshing here, and now my package is ready for download.

Review: Calling the Setup Script

Now which commands have we seen so far for the setup script? Well first of all, there's the sdist command which will create a source distribution, which is basically the simplest kind of package you can make. There's also another format, and we haven't seen that, but I do want to mention it here, and it's called a wheel package. And a wheel is a binary package format, you can make it with the bdist_wheel command for the setup. py script. Now setup. py doesn't by default support this format, but if you install a package called wheel, you will be able to create those. And the wheel format is in the process of becoming the de-facto Python packaging standards, and they're a little more sophisticated than a simple source distribution. One of the things they do is cache your installs for your whole system, which means that in many cases you get faster installation. And also if your project includes some C code, which will have to be complied in a platform specific way, will support it and you won't have to recompile so your installation will become faster and simpler. Now in many cases you can do without wheels, but once your projects get larger and more complex it is a good idea to look into this format. Then we saw that you can call the setup script with the register command to register your projects with the Python Package

Index. That will upload your project description to PiPy and you can then find the projects when you search the index. The next step is to actually upload your package. Now for a single release of your project you might have several packages. They will all be the same release of your projects, but one of them might be binary distribution for Windows, and another might be a wheel distribution for Linux, and a third might be a source distribution, etc, etc. In any case, you use the upload command for the setup script to upload these packages to the package index. And in the example here, we tell it to upload to the source distribution. So before you say upload in this command, you list the actual package you want to upload. And after that the whole world can find your project on the Cheese Shop.

Distributing Executables

Now there's a good chance that your project isn't just a library, but it also contains one or more scripts. And that means that you want the installer not just to enable people to import your Python packages, but you want to enable people to run your scripts. And there's a couple of solutions to handle this. The setuptools solution is to add an entry_points argument to the setup script. And as you can see the entry_points argument is a dictionary, and we'll give it one key value pair, and the key is console_scripts, and the value is a list of scripts we want to generate. So on install time, the setup script will generate executable files depending on the platform it's installing on. So in UNIX it will generate UNIX command line scripts, and on Windows it will generate Windows executable files. Of course you can add any number of scripts here, but I only added one for my demo scripts, and as you can see I called it amaze_demo, and the value I give it is the function I want the script to run when it's called. So the setup script will generate an amaze_demo file for me, I don't have to provide that myself, it will be generated and it will be put on the path so that people can run it. And when it's run it will call a function in my amaze. demo. tkdemo module, so that in my amaze package, in my demo package, in the tkdemo module, and then after a colon here, I say it's going to call the main function. So let's try this out, first of course I made some changes so I have to repackage. Very well. And now let's upload this to the package index. Now as you can see my upload fails because, of course, this package already exists, and because I made a new release I have to change my version number. So let's do that here, let's make a 0. 2 version. Very well. And now the upload works. Let's make a new virtual environment and see if we can install this. Ok, doing a pip install. Very well. It found and installed my 0. 2 version, and now you can see that it also installed an amaze_demo script. And like I said before, this is another file I made myself; the amaze_demo name is only present here in this string. So it's autogenerated. And I can very simply run it now here from the command line, like this. And there's my little demo user interface. So one of the ways to distribute executables from your project is to use automatic script creation by using the entry points setting in your setup. py. The big downside for this though is that people will already need a Python installation and they will need to know how to install Python packages. So what if you want to distribute something that's really to use by end users? So you don't want them to have to install Python, you just want to give them an. exe file, or a Mac OS app, or some other ready to use executable. Well there's several solutions for that. I'm not going to show you any of them, but I can give you some pointers. First of all, there's py2exe that creates a Windows executable for you from your Python code, and it will include a Python interpreter. So people won't have to install anything but your program and it will run without external dependencies. There's a similar tool for Mac OS. The Py2app tool will create a standalone Mac OS application. Again, it includes a Python interpreter so people won't have to install that themselves. And then finally there's another tool that's called PyInstaller, and that supports multiple platforms. So that's a single tool that will let you generate an. exe file and a Mac OS application, and more. So here you have several choices and your choice of tool will have to depend on your need for your projects.

Resources and Summary

So in this module we've seen a short overview of the tools there are to package and distribute your code, and how to use them. Now in case you want to know more, first of all here's the Setuptools Documentation. And I'm giving you a direct link here to the developer's guide for setuptools. Among other things you can find here a complete list of all the classifiers and all the options for the setup scripts. Another very valuable resource is the Sample Project for Packaging by the Python Packaging Authority. I already showed you this in a short demo and here's the actual link to the project. Finally, this is the tutorial from the Python Packaging Authority, which gives you the steps to setup your project for packaging, and how to run the setup scripts. So with that we come to the end of this module and the end of the course. In this last module we've seen how you can package and distribute your code using setuptools. I started with showing you how to prepare your projects for packaging, how to create a source distribution, and I also touched very briefly on the fact that there's another format called a wheel. Then we saw that with two simple steps you can upload your projects to the Python Package Index, and I gave you an overview of what to do in case you want to share your project as an executable. So in this course we covered a lot of ground. We looked at a lot of different aspects of the Python development cycle from setting up your environment to writing documentation, to packaging and distributing, and more. I really hope you enjoyed this course and I hope it helped you become a more professional python developer.

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	Intermediate
Rating	★★★★☆ (280)
My rating	★★★★★
Duration	2h 19m
Released	18 Jul 2014

Share course