Getting Started with Jupyter Notebook and Python

by Douglas Starnes

Start Course

Bookmark Add to Channel Download Course

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

# Course Overview

## Course Overview

As one of the most versatile and popular programming languages, Python wears many hats in software development, and such a diverse set of use cases requires a diverse set of tools. I'm Douglas Starnes. I've been using Python for almost a decade and have deployed solutions in academia, industry, and government. I'm a frequent speaker on Python and related topics. I also help run a Python user group, and I just created this course, Getting Started with Jupyter Notebook and Python. In this course, you'll meet Jupyter Notebook, a unique development tool that takes the capabilities of the Python rebel to exciting new heights. I'll show you how easy it is to get Jupyter Notebook up and running, how to be productive with shortcuts, magic commands, and the integrated help. We'll see how images, visualization, and rich text can help you communicate your ideas very effectively. We'll also look at widgets that make Jupyter Notebook even more interactive by developing user interfaces and dashboards inside of the notebook. And then we'll see how you can use Jupyter Notebook to collaborate and share with others. Finally, I'll show you the next step in the evolution of Jupyter Notebook. Thank you for watching the course, and I look forward to helping you become a Jupyter Notebook ninja.

# Installing Jupyter Notebook

## Installing Jupyter Notebook

Welcome to the exciting world of Jupyter Notebook. If you've never used a tool like Jupyter Notebook before, this is going to be a real eye-opener. In this module, we'll talk about what Jupyter Notebook is, where it came from, why it's important, and how to get it up and running. Jupyter Notebook is the tool for data scientists, and while it has a lot of practical use in other areas as well, you'll find it near the top of every data scientist's list of favorite applications. But Jupyter Notebook is a different kind of tool. If you come from a more traditional development experience, you're used to an IDE, or Integrated Development Environment, such as Visual Studio or Xcode, or a lightweight text editor like Visual Studio Code or Atom. Jupyter Notebook is like none of these. Instead, it is an interactive environment, and we'll see more about how this differs from an IDE in a minute. First, I want to point out the workflow, which is the inspiration behind Jupyter Notebook. Data scientists will often work on isolated problems as opposed to whole applications, thus they will set up an experiment with an environment specific to that problem. To prepare the experiment, they provide it some initial parameters and input, or well-known values, in order to run the experiment for the first time. The experiment will yield some sort of results, which may or may not be what was expected. Usually this will require incremental adjustments to the parameters after which the experiment will be run again, and this loop will repeat until the desired results have been achieved. And while this is possible using a text editor and command-line compiler or interpreter, or even an IDE, Jupyter Notebook is designed for this kind of iterative development. If you've ever done any front-end web development, you've likely used the Chrome Developer Tools or the F12 Developer Tools in Microsoft Edge. In both, there's a JavaScript console that allows you to execute statements in isolation as opposed to placing all of your code in a file and then executing the entire file at once. This is known as a REPL, which is an acronym for Read, Evaluate, Print, Loop, that enumerates the steps to provide this interactive experience. Python, the language that we will use for this course, also provides a REPL, which is even more bare bones than the one in Chrome Developer Tools. These shortcomings were realized by Dr. Fernando Perez, a physicist and statistician at the University of California at Berkeley. To combat this, he created IPython, the enhanced Python interpreter. This extends the REPL with useful features such as syntax highlighting, automatic indentation, the ability to execute shell commands, special and extensible magic commands, and much more. While IPython was a great leap forward, it still had a few pitfalls. One of these was collaboration. Scientists not only work on isolated problems, but they work on them with other scientists, so they need a way to communicate the progress and results of experiments. This is difficult to do with IPython because just like the Python REPL, once the application is closed the session is gone, there is no way to persist it. This was resolved with IPython Notebook. IPython Notebook starts a local web server and runs a Python session that is rendered in a web browser. These sessions, called notebooks, are persisted to disk and can be shared with others. In fact, you can push an IPython Notebook file to GitHub and it will render a static version. Also, scientists love visualizations. IPython Notebook will render visualizations inline and persist the contents in the notebook file. And when a scientist describes the experiment they are working on, comments in the code just won't cut it, so with IPython Notebook they can insert rich text using Markdown, and that is saved as well, and it's rendered in GitHub too. Now you may have noticed that I've been talking about IPython Notebook and not much about Jupyter Notebook. That's because in the beginning there was only IPython Notebook, but data scientists use languages other than Python. They might use R, which is another popular choice for data science. If you're on the. NET side, you might have heard of F#. These other languages saw IPython Notebook and wanted to play too. The nonlanguage-specific features of IPython Notebook such as inline visualization, file persistence, shell commands, magic commands, and more are relevant to these other languages as well. So the IPython team abstracted those features from IPython Notebook and made it possible to plug in languages and get the same development experience that IPython developers were using. This shell that serves these features was then renamed to Jupyter Notebook. So while this course will focus on Python and Jupyter Notebook, keep in mind that the only thing specific to Python is the code. Many of the Jupyter Notebook features covered will carry over to other languages as well. To get all of these cool new tools, all you have to do is download them. Just like Python, Jupyter Notebook is free, open source, and cross platform. It's hosted on GitHub, so you can hack on the source and suggest new features. To install it with Python, just use the pip package manager. However, the way that I recommend installing Jupyter Notebook locally is with Anaconda. This is a free data science distribution, which can be downloaded at this URL. Anaconda contains over 1000 data science packages including Python, IPython, and Jupyter Notebook. It's quite a large install at around 400-500 MB, but it's a one-stop install that gets everything you need for this course and beyond. Anaconda is also cross platform and is based off of open source tools as well. If you want to run Jupyter Notebook locally, but don't want to install anything that might interfere with your local machine, you can use Docker. There are several Docker repositories which contain the Jupyter Notebook preinstalled. The Jupyter project itself maintains a repository, and Anaconda has one too. There are a few nuances to getting Jupyter Notebook up and running with Docker, and I'll mention those during the demo. Finally, if you don't want to install anything at all and still not pay anything, you can use Azure Notebooks, which is Jupyter Notebook as a service. The only thing you need to do is go to this URL and sign in with a Microsoft account. If you don't have one, just sign up for a free outlook. com email address and you'll be ready to go. Azure Notebooks will let you run Jupyter Notebooks in the cloud. It comes with many of the popular data science and machine learning libraries preinstalled, including NumPy, pandas, matplotlib, TensorFlow, and Keras. You can also share and publish notebooks similar to how you would on GitHub, except that they are live, not a static rendering of the content. And we'll see more of this in the demo too. Speaking of the demo, that is coming up next.

## Demo: Installing Jupyter Notebook

In this demo, I'm going to show how to make sure that your Jupyter Notebook installation is up and working correctly. First I'll discuss the local installation via either pip or from Anaconda. I won't show the actual installation because it varies between platforms; however, it should be self-explanatory to anyone who has used Python before. After the installation, open a terminal or command prompt window, depending on your operating system, and simply run the command jupyter notebook. This is will start the Jupyter Notebook web server on port 8888 on the local machine. In most cases, it will also open the default web browser on the system to localhost on port 8888. If it does not, simply copy and paste this address into a new browser window. Notice this token at the end of the URL. This is a security feature as Jupyter Notebook allows shell access to the entire machine running as the user, which started the notebook server. The security token prevents just anyone from logging into your notebook server. To stop the notebook server, I'll press Ctrl+C twice. There are several options that you may find useful when starting the notebook server. The first is the --port option. If port 8888 is already in use, then you can specify a different port by passing it to the --port option. There's also a --ip option, which let's you specify a particular IP address to listen on instead of the default local address. By default, the server will serve files from the directory in which it was started; however, you can use the --notebook_dir to use any directory. And finally, if you don't want the web browser to open automatically in the case that the server is running remotely, you can pass the no-browser option, which doesn't take any arguments. Next, let's look at Docker. Since this is not a course about Docker, I'll assume that you already have it installed, but the process is not too hard, basically just download the installer and run it. After that, the Docker command-line interface will be available from the terminal or command prompt. Next, I'll pull a repo from Docker Hub. Here I'll just get the Jupyter image for the data science notebook, which has a lot of the libraries that we'll use later in the course preinstalled. Now that the image is downloaded, I can create a new container with the docker run command. The -it option tells Docker to open the terminal for the container where we can run commands and view output. The --rm command tells Docker to remove the container instance once we exit. Finally, jupyter/datascience-notebook is the name of the image to create the container from. The notebook server will start automatically and give me the URL with a security token; I'll copy and paste that into a browser. And we get an error. That's because the server is running inside of the Docker container. To access it from the host operating system, we need to map a port from the host OS to port 8888 in the container, and this can be done with the -p option. So I'll go back over to the terminal, press Ctrl+C twice to exit the server, and automatically remove the container, then I'll modify the docker run command. Before running this, there is one more option that can come in handy. Any notebooks created inside of the container will be stored in the container, and thus destroyed when the container exits. However, we can map a volume on the host OS to a volume in the container and share storage between them. The -v option of the docker run command handles this. The first path is for the host OS and will differ depending on what host OS you're running. The second is for the container and will always be a Linux style path. Note that also you may have to give Docker permission to share the drive for the path of the host OS, and this can be done in the Docker settings. After Docker restarts, the command should run just fine. I'll copy the URL once more, and the notebook server is up and running. Finally, let's take a look at Azure Notebooks. For this, I don't need to download or install anything, all I need is a Microsoft account, which for most people is an outlook. com email address. So, I'll go to a new browser and navigate to notebooks. azure. com, and then I'll sign in. On Azure, notebooks are stored in libraries, so first I'll create a new library by clicking on this large plus icon. I'll give the library a friendly name, and also an ID that will be part of a unique URL for the library. The checkboxes I'll leave at the default values, then I'll go to my new library, create a new notebook, give it a name, and then I can choose what type of file and what language to use. Notice that I have 3 choices here for versions of Python, 2. 7, 3. 5, and 3. 6, and I can also use R and F# as well. Additionally, I can create blank files for documentation, data, and other things. Note that I can also upload files from my local computer and fetch data from a URL, but for right now I'm just going to create a Python 3. 6 Notebook. The new notebook can be seen in the library, and clicking on the link will open it. Now this may take a few seconds as Azure spins up a new notebook server. It's worth pointing out that since Azure Notebooks are a shared resource, they will time out if you don't use them for a while. The notebooks and other files you create are still saved, but you might have to reopen them to restart the notebook server. You can use any of the methods discussed to follow along with this course. I will be using a local installation with Anaconda because it's safer and doesn't assume a working network connection. In module 2, we'll see the fundamentals of working with a Jupyter Notebook. We do some ad hoc data analysis to see how it works. After that, you'll begin to see how it can make your life much easier.

# Moving from the REPL to a Notebook

## Creating a Jupyter Notebook

In this module, we will get our first hands-on exposure to Jupyter Notebook using Python. We'll look at the basic features of using notebooks and simple data analysis. Then in future modules, we'll build on these concepts. And that's it for the slides, because the best way to learn Jupyter Notebook is to see it in action. So I'll start off by firing up the notebook server in the terminal. Again, I'm using Anaconda, so I'll my default browser to this page, which is a listing of the directory in which the server was started. The entries in the directory are actually hyperlinks, so if I click on a folder, it will switch to a listing of that folder, and if I click on a file, it will attempt to display the contents of that file. And as I navigate the file structure, the current location is tracked using this breadcrumb menu. I can click anywhere in it to go to a particular folder, and to get back to the root click on the folder icon. With the new menu in the right corner I can create new files, folders, and notebooks. I'll discuss the terminal in the next module. Depending on what is installed for your system, there could be other languages for notebooks. For now, I'll create a new Python 3 Notebook. This is the interface for Jupyter Notebook. Among the things I might want to do right away is change the name of the notebook. So I can just click in the title and type a new name in the dialog, and now if I look back at the directory listing, I can see that the new notebook file has been saved. This is the interface for Jupyter Notebook. The area with the green border around it is called a cell, and cells contain different types of content in Jupyter Notebook. One type of content that it can contain is source code. For example, I'll get the obligatory variation on hello out of the way. Two convenient things happened here. First, Jupyter recognized the print function from the Python language and highlighted it. Also, when I type the opening parenthesis, it added a closing parenthesis. These features are like what you would find in most modern text editors. It will also add closing quotes for me automatically. Now to run the code in this cell, I can go to the Cell menu and click Run Cells, which will execute the code in the currently selected cell or cells. This will run the code in the cell and print the output, if any, below the cell. There are also several keyboard shortcuts you can use to execute cells. Over time, it will be come obvious that using the keyboard with Jupyter Notebook is much quicker than using the mouse, so the shortcuts are well worth learning. I'll save a detailed discussion for later in the course, but the keyboard shortcut I use most often to run a cell is Shift+Enter. The output of the cell did not change; however, notice the number next to the cell, it changed from 1 to 2 indicating that the cell was run again. Jupyter Notebook also offers other text editing features that you would expect to find in most modern editors. Context-sensitive suggestions is one of these. First, I'll import the OS module from the Python standard library, which exposes an API for working with the file system. If I want to list the contents of the current directory, I can call the listdir function with no arguments. To speed things up a little, I can just type the first few letters of the listdir function and press Tab to complete it. If there is more than one choice available, Jupyter Notebook will display a drop-down, which will narrow in on a set of options as I type the name. There's a subtle exception to this rule. Jupyter Notebook can only infer about code that has been executed in cells, so in order to get the tab completion, some other code may need to be executed first. For example, I'll import the datetime module; however, I won't execute the cell and instead enter code that tries to use the datetime module in the same cell. Even though I'm pressing Tab, I'm not getting any suggestions, and there are plenty of possibilities to complete this statement in the datetime module. This doesn't mean there's an error in my code or a bug in Jupyter Notebook, it's just that the cell with the code which imports the datetime module hasn't been executed yet. Now I can still type out the code manually and it will work just fine, and in future cells I'll get tab completion again. So if you aren't getting tab completion in Jupyter Notebook, it does not mean that you have an error in your code. Jupyter Notebook is not an IDE, it's an interactive computing environment, so the rules are a bit different at times. Here's another neat tip related to tab completion. The initializer for datetime takes a number of required and optional arguments and in a specific order, so it can be tough to remember them all. With my cursor over the datetime initializer, I can press Shift+Tab and a pop-up will show the signature of the function. This is part of the integrated help in Jupyter Notebook that I will cover in more detail in the next module. One really nice thing about the tab completion is that it works with your code as well. For example, I can store the output of the listdir function from above. Now the files variable is available in tab completion, and Jupyter Notebook knows that files is a list, so I can get tab completion on that as well. Likewise, if I were to write a class, just like with variables, I'll get tab completion on the class name. I'll also get the Shift+Tab pop-up help, and I'll also get tab completion for the class members.

## Managing Groups of Cells

Any kind of Python code can be run in the cells. Here's a function that will generate prime numbers using the sieve of Eratosthenes. Notice that Jupyter Notebook automatically indents the code where required. It's not necessary to understand the innerworkings of the algorithm for this exercise, just know that it will generate prime numbers up to and including the value passed to the function. I'll execute this cell with Shift+Enter to make the function available in the rest of the notebook, and now I can call the function. Now I've changed my mind. Instead of printing the primes, I'd like to have them returned in a list. I can simply click on the cell and make the necessary changes. Let's see how it works. Hmm, I'm still getting the same input as before. This is a common mistake when first using Jupiter Notebook. If a cell has later cells that depend on its value, the dependent cells will not automatically be updated when the original cell is modified, so I need to select the cell with the function and re-execute it. Notice how the cell that called sieve is now selected? This is the other part of what Shift+Enter does, select the next cell or add a new one if the selected cell is the last. So now, I can just press Shift+Enter again, and there's the list. Alternatively, there are some menu commands which could help out in such a situation. To demonstrate this, I'm going to open up a notebook that I created ahead of time. This notebook has some simple text analytics in it. What exactly it does is not important, but many times you'll be in a situation where you could have been handed the premade notebook and just want to get to the end result. It is possible to repeatedly press Shift+Enter until you get to the end of the notebook, but Jupyter offers a better way. In the Cell menu is an option to Run All. This will run the cells in a notebook from start to finish, and we can see that in the provided text each contains on average 12. 75 letter T's. Suppose that now we need to consider new lines in the text, so I'll add the new line as \n to the list of valid characters in the strip_non_alpha function. I'll also add a new cell with code to count the new lines, which only exist in the last text. Now I only need to run from the second cell on, so I'll select the second cell by clicking on it, and then I'll select the menu option Cell, Run All Below, and the nursery rhyme in the last text has five new lines. Note that the name Run All Below might suggest that the selected cell is excluded, however, the option does execute the selected cell and all of those following it. This was a simple example, but the technique has practical applications. Consider the scenario of something like a machine learning experiment. These can take a long time to run. If all you want to do is work with the results, it's not always necessary to run the entire experiment each time you compute a new statistic. Or what if you were getting data from an API that is write limited? You can run just the code that works with the data after it has downloaded instead of making a wasted call to the API. This would be extra work in the edit and compile workflow, but Jupyter Notebook includes it automatically.

## Shell Commands and Special Objects

In addition to executing Python code, there are other types of actions in Jupyter Notebooks. One of these is the ability to run shell commands from within a notebook cell. To start off, I'll open a new Python 3 Notebook, and then I'll change the name to shell_commands. In order to execute a shell command inside of a Jupyter Notebook, it must first be prefixed with a bang, or exclamation point. This will help distinguish it from Python statements, and as we will see later, magic commands. Let's start off with something simple. The shell command pwd is short for print working directory and will return the absolute path of the current directory when I press Shift+Enter. I could also use the ls command to list the contents of the current directory. Just like at the terminal, shell commands can take options and arguments. (Working) I'm working on a Mac right now, which runs an operating system with a variant of UNIX from which the ls and pwd commands are inherited. Windows is a different story. These commands and many other UNIX commands do not exist on Windows. For example, on Windows you would use the dir command in lieu of the ls command. And this brings up another good point about shell commands. While Windows has the dir command, dir is also the name of a built-in Python function. When called, the dir function in Python will list the names of the Python module's functions and variables that are currently in scope. The moral of the story here is to make sure to prefix all shell commands with a bang. Now you may accidently discover that some of them work without the bang, and this is not an oversight by the Jupyter team, it's actually by design. However, for the purposes of this course, opening the bang can get really confusing. To keep things all on the same page, I'll be using the bang for all shell commands. So far, this might not seem like a big advantage, especially when you have to take all of the platform peculiarities into consideration, but with Jupyter Notebook you can do more with shell commands than simply display their output. The output of a shell command can be captured and used with Python code. At this point, files is a list of the names of the files and directories in the current directory. So now, I could use it as regular Python list. In addition, I can also use Python variables in shell commands. For example, in this directory I have a file named README. txt. As a reminder, the cat command in this case simply dumps the content in a file to the terminal. If I store the file name in a variable, I can pass it to the cat command by surrounding it in curly braces and and the value will be substituted by Jupyter Notebook. We're beginning to build up a lot of cells, and it could useful to replicate the contents of the cells and the results somewhere else in the notebook. To do this, Jupyter Notebook stores all of the inputs and outputs into objects creatively named in and out. You've likely noticed the numbers next to the inputs and outputs of the cells in a notebook, which increment each time a cell is run. These numbers are indices to the in and out objects for that particular input or output.

Let's take a closer look. First, I'll import the Counter class from the collections module in the Python standard library. The counter is a dictionary-like object which keeps counts of occurrences of different keys. Here's another place where tab completion comes in handy when importing modules, as well as the members of modules. Note that the cell has the In object with a specific index for the import to the left. Now I can reproduce the input by accessing the In object. It might be more useful to replicate the output. I'm going to use the Counter class to count how many words begin with each letter in this text, then I'll get the count for the word with the letter t, and there are 8 of them. In 14 is again the content of the input or the text of the cell. Out 14 is the output. The input is going to be text, but the output can be any valid Python value, and thus can be used in future cells. If In 14 contained a lot of code, this could be a big timesaver. Notice that not all cells have output. The cell with the import did not produce any output, so trying to access that index will result in a KeyError. This also works for shell commands; however, they don't produce any output. One last thing in this video. The underscore character has special meaning within the Jupyter Notebook environment. It will always store the most recent output, whatever that may be. So if I run this code, the output will be a list of the squares and the integers up to 10. This will also be the value of the underscore. And if I do something different in the next cell, the output changes, and thus the value of the underscore is updated as well. Note that the underscore has the value of the most recent output, not the output of the last cell in the notebook. If I execute the first cell again, and then get the value of the underscore, I get the list again. This is really useful when cleaning up text using regular expressions. The re module in the Python standard library will handle this. While a complete discussion of the module is outside of this course, it is enough to know that the sub function will take a regular expression, a replacement string, and target string, then any text in the target string matched by the expression will be replaced by the replacement string. So let's say we have some text like this. Cleaning up this text might involve compressing the whitespace characters and removing all of the nonalphabetic characters. Regular expressions can handle this. First, I'll import the regular expression module, then I'll remove all of the digits and punctuation. Now I want to remove consecutive whitespace characters, replacing them with a single space. I'll use a second regular expression, this time with the output of the previous one. So this works, but did it really require a lot of extra effort or code? No, not in this case, but imagine you have a really long chain of regular expressions. Keeping up with all the temporary variables could be confusing. Using the underscore character makes it simpler and cleaner. Here I arrived at the exact same result, but without using temporary variables. Instead, the second call to the sub function simply used the underscore as input because it is the output of the previous line. Now one word of caution about using the underscore character. Often it is common to use the underscore to represent a throwaway variable in Python. For example, if I de-structured a tuple and only wanted to use the first and third values, I might do something like this. Now protocol is https, port is 80, and the underscore is Pluralsight. com. However, the value of the underscore was explicitly assigned in code as opposed to implicitly getting the value of the most recent output. In this case, not only is the value of the underscore clobbered, but the functionality of it is too. So in the future, the value of the underscore will not hold the most recent output. In this case, I would expect the value of the underscore to be hello world, but since I assigned its value in the tuple de-structuring, its value no longer contains the most recent output, but instead the most recent explicit assignment. Another time this might trip up your code is in a loop when you want to show that the value of a counter variable will be not used. Again, the functionality of the underscore is specific to Jupyter Notebook, which was inherited from IPython, so it works there too, but it won't work in this Python interpreter.

## Styling Cell Output

We've seen how Jupyter Notebook can execute and format code in a cell, and then we saw how you can execute shell commands and capture the output in several ways, but there is more that you can do with cells. There are a number of libraries, which when used inside of Jupyter Notebook render the output of a cell in special ways. In this video, I'll take a look at one of them. We're going to need some data to work with. I have a simple comma-separated file on GitHub with precipitation data about the largest cities in the state of Tennessee. To make sure it is accurate, I scraped it from Wikipedia. To get the file onto my local machine, I can run the curl command using the technique in the previous video for executing shell commands; and here's the file. Notice that with Jupyter Notebook, we have tab completion for the Python variable names and shell commands, as well as path names too. Python provides a csv module in the Python standard library. It's simple enough that we can use it here to load the csv file. The reader inside of the csv module will take the file handle and use it to read the contents of the csv file. This works, but it's kind of hard to get much out of this by looking at it. The csv module is intended for use in code and does not worry much about display. Meet pandas. Pandas is a library for data analysis. It mimics a lot of the functionality available in R, so you get the best of both worlds; the power of R without the quirks of R. While a complete discussion of pandas is outside of this course, it's enough to know for now that pandas provides a two-dimensional data structure, the data frame, which is analogous to R's tabular data structure of the same name, into which we can load data and work with it. So, I'm going to start off by importing the pandas module. Using the pandas read_csv function, I can import it into a data frame. Again, we get tab completion for path names inside of Jupyter Notebook cells. This can be a real timesaver when you can't remember the name of a file or trying to type out a complex path. Now I'll just show the data frame. Wow, this is already better. Pandas has told Jupyter Notebook to use rich formatting to display a data frame. It has recognized column headers, spaced everything to align the values, used a special value for missing values, numbered the rows, and also, as you hover over the table, the rows are highlighted, and all of this just for showing up. So it turns out that the rendered table is styled using HTML. We can access the styles via the style property of the data frame. The style property has some default functionality built in as well. For example, it can highlight the maximum value in each column; it can also do custom highlighting as well. Let's highlight any value, which is 5 or greater, by making the text green. First, I need to write a function that will take in a value and return a CSS style based on the value. Columns in pandas are typed, so the first two are strings, actually objects in pandas, while the rest are floats; therefore, I have to check the value as a float or otherwise I'll get an error that floats and strings cannot be compared. Then based on the value, I'll return a CSS style for green if the value is a least 5, and black if it's not. All of the strings will continue to be black. The applymap method on the style property will take the function and apply it to all of the values. I can also use a simpler function that does not check the type of the value by telling the applymap method to consider only a certain subset of the data frame. Since the columns from the second index and higher will only have floating point values, I don't have to check the type anymore. Pandas gives you very fine control over the formatting of data frames with CSS. If I wanted to change the colors of the rows in the data frame, I could add a style element with the HTML type. I've got a CSS file named ugly. css because I am not a web designer, so prepare your eyes. Here are the rules. The tables rendering a data frame have a class of dataframe. After that, I just access the tags inside of the table and use CSS rules to change the background colors. Next, I will import the HTML type from the IPython. core. display module. This will insert and render HTML into the notebook. I can also use HTML to insert a style tag to render the CSS rules. We'll revisit formatting and displays in a later module. But coming up next is a survey of other Jupyter Notebook features. These will make your life easier and your work more productive.

## Leveraging Special Notebook Features

## Keyboard Shortcuts and Inline Help

This module is dedicated to the features of Jupyter Notebook that will make your life easier. Knowing these features is not a requirement to use Jupyter Notebook, but doing so will afford you a more productive experience. There are a number of keyboard shortcuts provided by Jupyter Notebook, and I'll go through these fairly quickly. First of all, a list of keyboard shortcuts doesn't mean much without seeing them in context, and also I'll be using these through the course, so there will be future opportunities to discuss them. Jupyter Notebook also provides a unique help system, which is context sensitive yet unobtrusive at the same time. Now we've already seen how to execute shell commands from within Jupyter Notebook. In this module, we'll see magic commands. Magic commands wrap functionality specific to Jupyter Notebook, many are included by default, and you can even make your own. Finally, I'll discuss some security concerns that you need to be aware of, especially if you consider making Jupyter Notebook available on a server. I've divided the keyboard shortcuts into those available in command mode and those available in edit mode. Again, edit mode is when you are able to modify the content of cells. Manipulating the cells themselves happens in command mode. Let's start with command mode. Cutting the selected cell has a similar shortcut as Windows or macOS, except instead of using a modifier key such as Ctrl or Cmd, just use the X key alone. Copying the selected cell is the same except using the C key. The shortcut for paste is V, except the behavior is a little different. Since we are working with cells and not text, we have two places to paste a cell, either above or below the selected cell. The V key by itself will paste below the selected cell. To paste above the selected cell, add the shift modifier to the V key. To delete a cell, press the D key quickly twice; this shortcut will be recognized by Vim users. Finally, to insert a cell we have a situation similar to pasting, we have to decide whether to insert above or below the current cell. Now these shortcuts are much simpler to remember. Use the A key to insert a cell above, and B to insert a cell below. So that's A for above and B for below. If you have a cell selected in command mode, you can extend the selection by holding down the Shift key and using the up or down arrows. You can also use the mouse, though you'll find that using the keyboard is much faster for most situations. We've already seen how to run cells in a notebook, but there are some subtleties that are worth pointing out. Ctrl+Enter will run the selected cell and leave that cell selected, so Ctrl+Enter will not change the cell selection. And this is useful if you want to see the output of success and executions of a cell. Shift+Enter, as we've already seen, will also execute the current cell. However, it will also select the cell below unless the selected cell is the last cell in the notebook in which case it will insert a new cell below and select that cell. Alt+Enter also executes a selected cell and then always inserts a new cell below. Most of the time I use Shift+Enter, but Ctrl+Enter comes in handy. Now after you have a set of cells that works how you want it, it can tedious to keep using Shift+Enter to run through them, and this code might be better combined into one cell. Before you start to copy and paste, Jupyter Notebook has thought ahead. Pressing Shift+M will combine or merge all of the selected cells into a single cell. After you merge a collection of cells, the result could have a lot of code in it and line numbers would help. To turn on line numbers, press Shift+L; to turn them off again, press Shift+L again. That's all the command mode shortcuts I want to point out. Now let's look at some edit mode commands. To cut, copy, or paste text within a cell as opposed to the actual cells themselves, just use the shortcuts available in the browser. So on Windows this would generally be Ctrl+X, C, and V for cut, copy, and paste, and macOS would be the same except use the Cmd key. I'm sure that you need to reminder of the significance of whitespace or indentation in Python; therefore, it should come as no surprise that Jupyter Notebook has keyboard shortcuts to indent and dedent code. Finally, we saw earlier that it is possible to merge a set of selected cells. It can be also practical to split a large cell into multiple cells. Inside of a cell in edit mode, pressing Ctrl+Shift+Minus will split the current cell into two at the cursor. This is helpful if you are running Jupyter Notebook on a server with shared resources and there's a quota for compute power. Splitting a large cell into multiple cells might help you stay within the limits by distributing the work over multiple cells instead of trying to do it all at once in a single cell. Now those are all the shortcuts I'm going to cover specifically, but if you want to see the entire list, just press the H key while in command mode, so H for help. Speaking of help, Jupyter Notebook also provides a convenient inline help feature. This is inspired by IPython and uses the same syntax. Appending a question mark to any member in your code will bring up help for that member when the cell containing it is executed. Now this help is actually the docstring for the code. Docstring is a string literal in the first line of a Python module, class, or function that describes it. So it's kind of like a comment that the user can access. The docstring will appear in a pane that will pop up at the bottom of the window. This way you can still work on your code while the help is visible, and then dismiss the pane when you no longer need it. Also, if the pane shows no help, it's because there is no docstring for the specified member. Docstring is not a requirement, it's just a suggesting like PEP 8. In fact, docstring itself is a PEP, PEP 257, so not all code will use Python docstring, although it probably should.

## Demo: Keyboard Shortcuts

The demo for this module is split over the next two videos. In the first, I'll discuss the keyboard shortcuts in the help system. In the next one, I'll look at the magic commands and show why Jupyter Notebook needs to be secured outside of a local instance or server. So I'll start off in a new Jupyter Notebook and I'll press Enter to switch this cell to edit mode, then I'll import the random module from the Python standard library. Now I'll press Shift+Enter to execute the cell and select the next cell, but since this is the last and only cell in the notebook right now, this will insert a new cell at the end of the notebook. Now I'll use the choice function in the random module to select from a list, and I'll use a list of even integers. I could also execute this cell with Shift+Enter, but I would like to see what happens if I run this cell several times. So instead I'll use Ctrl+Enter, which will execute the cell, but not move the selection. And if I repeat that several more times, I can see that this code is working as expected. Now the random module might also be helpful for generating passwords by selecting random characters. The string module from the Python standard library has members that include different sets of characters such as uppercase/lowercase letters, as well as digits and punctuation, all of which should be included in a strong password. So I'll import the string module, but don't touch that mouse. Instead, I'll press the A key to insert a new cell above the selected one. To execute this cell, I'll press Alt+Enter to insert a new cell below the selected one after it is run. Remember that Alt+Enter always inserts a new cell where Shift+Enter will only insert one if the selected cell is the last one in the notebook. In this new cell, I'll concatenate all of the characters that are possible in a password. I'll use Shift+Enter to execute this cell and select the one below. In this cell, I'll modify the call to random. choice to select a list of random characters and join them together. I can run this cell several times with Ctrl+Enter, and after a while I notice a problem. The frequency of the different sets of letters is inconsistent, some are majority letters and some are more punctuation than others. This is because there are many more letters than digits, and there is more punctuations than there are characters. So I'll need to create a different haystack with more equal frequencies. For the purposes of this demo, I'll just repeat or truncate the different sets. I need to modify the haystack, so I just need to copy that cell. Since I'm already in command mode, I'll press the up arrow to select the cell with the haystack, then I'll copy it with C, and paste it below with the V key. To paste above, I would use the shortcut Shift+V, and now I can create a new haystack. I'll press Shift+Enter to execute this cell, and then C and V again to copy and paste the cell with the join function, then modify the copy to use haystack2. Now I'll run it a few times and it's better, but close enough for demo work. Now I want to delete the cells containing the code for haystack. I'm already in command mode, so I can select this cell here by pressing the up arrow and delete it by pressing the D key twice, and I'll do the same for the other haystack cell as well. Now this is working well, however, my code is spread out over multiple cells, so I want to merge them into a single unit. I'll select the first cell, and then while holding the Shift key I'll press the down arrow to select the rest of the cells. Finally, I'll press Shift+M to merge them. Next, I'd like to put the random code into a function, so I'll add a def statement. The body of the random_password function needs to be indented, so I'll select it and then press Cmd+right square bracket to indent; on Windows I would use Ctrl. Using Cmd or Ctrl+left square bracket will de-indent. While I'm here, I want to change haystack2 to haystack. I can change both of these lines at once by holding down the Alt key and selecting the columns in the code. Now I can just delete the 2. Of course I'll need to update the random. choice function as well. Finally, the range needs to take the linked I and return the joined list. Now I can execute this cell and call the random_password function in a new one. One last thing. Let's say we want to pass the haystack into the random_password function. For testing, it would be useful to isolate the function in a single cell, so I'll put my cursor above the def statement, and then press Ctrl+Shift+Minus to split the cell. Now I can modify this function to accept the haystack. Finally, by pressing L in command mode, the cell will toggle the appearance of line numbers.

## Demo: Inline Help

Pythonistas are always looking for a way to build a better mousetrap, and it's even better if that mousetrap has already been built. With this in mind, perhaps there are some functions in the random module that can make the password generator better. I'll start off by getting a listing of the random module with the dir function. Here's one that looks promising, shuffle. I wonder what it does? Using the built-in help with Jupyter Notebook, I can get a summary of the function by appending the question mark to it and executing the cell. This pop-up at the bottom of the screen slides in. The shuffle function will rearrange the elements of a list in place, not quite what we're looking for because we need to generate the list. I'd like something that will select all of the password in one shot instead of using a list comprehension. How about sample? Chooses k unique random elements from population, which would be the haystack from above, and this could work, but do we really need to enforce the unique requirement? How about choices? Ah-ha, this one returns a k sized list from population, but with replacement. There are some other keyword arguments here that assign weights, which we could possibly use to avoid repeating or truncating elements in a haystack, but omitting those will do the exact same as it was comprehension above. Now I can modify the random_password function to use choices instead. Now all I need to do is have the random_password function join the result of random. choices. The built-in help also works outside of the standard library, and you can even take advantage of it with your own code. First, I'll try to get the help for the random_password function. Not a lot here. Notice the docstring, it's empty. Docstring is like inline documentation for Python code. The way to create docstring is including a triple quoted string as the first line of a module class or function. I'll query a simple docstring for the random_password function. An extensive review of docstring conventions is beyond the scope of this course; check out PEP 257 on python. org for more details. However, after executing this cell again, I can now get help for the random_password function. In the next and final video of this module, we'll see magic commands.

## Magic Commands and Security Concerns

Magic commands are another useful Jupyter Notebook feature. Like the help system, magic commands are inspired by IPython, and in fact the commands are largely the same in both. Magic commands are similar to shell commands as they may take arguments or options, but they are specific to IPython and not Python statements. As we will see later in the course, they may even be written in Python. They are always prefixed with the percent sign to distinguish them from shell commands and Python statements. If you need help for a magic command, append a question mark to it and execute the containing cell, just like the help system we saw earlier. Now there's a long list of magic commands in the IPython documentation online, but I've listed a few of them here. The lsmagic command simply lists all of the available magic commands. The history command will let you manipulate the session history, similar to the history in the command line in Linux; however, this command is specific to the Jupyter Notebook session. The load and run commands let you execute Python code from external files. Jupyter Notebook also supports aliases via magic commands. And if you need to start all over, the reset command will restore the default settings in the notebook. And I'll go over these and a few more in more detail in the demo. Before the demo, I want to make you aware of a few security concerns that Jupyter Notebook presents. Remember that by default Jupyter Notebook allows unrestricted access to system resources, and it is very easy to put a Jupyter Notebook server on the web. These features combined create the potential for a security nightmare as you can run shell commands from within a notebook. Adding to this, some installations such as Anaconda allow you to open a terminal emulator in a web browser, and it's much the same as if you had an SSH connection to the machine. Now, if you were running the server locally or in a Docker container this isn't a concern, but if you were considering a virtual machine in the cloud as a notebook server for convenience, it will need extra configuration to be secure, otherwise a rouge user could do damage to or with the machine it is running on, and the security token is not sufficient for this scenario. Also, it may be convenient to run a Jupyter Notebook server for a workshop or training session. In this scenario, multiple users would likely be accessing the same server. However, by default, Jupyter Notebook is not meant to be multiuser. In other words, the users would all be competing for the same resources, so a change or a mistake by one would be seen by all. There are ways to lock down Jupyter Notebook and make it support multiple users, but those are beyond the scope of this course. I would recommend looking into JupyterHub to learn more. I would add that for many cases in which you want to run Jupyter Notebook remotely or have a multiuser environment, Azure Notebooks and the Anaconda Cloud are great places to get started and will handle a wide range of needs.

## Demo: Magic Commands

Here I've got a blank notebook and I'm going to start out by running the ls magic command. So I'll put %lsmagic in the cell and press Shift+Enter to execute it. The output is a listing of all the magic commands available on this notebook server. Notice that they are separated into two groups, line magics and cell magics. The difference between the two is subtle, but important. A line magic is very similar to a shell command in that all of its input will be contained on the same line as the command itself. A cell magic will also take the rest of the line as input, but also the rest of the cell as well. Cell magics are easily identified because they are prefixed with an additional percent sign. Let me show what I mean. One of the commands I didn't mention in the slides is called writefile. Writefile is a cell magic command. Consider this simplified version of the random_password code from earlier in this module. If I want to make this available outside of the Jupyter Notebook, I can write it to a file with the write cell magic. The write cell magic will take the name of the file to write as an argument on the same line, but it will also take as its input the contents of the rest of the cell, which will be written to a file. So I'll prepend the magic command to the cell, I'll execute this cell with Shift+Enter, and get the message that it is writing the utils. py file. Now I can run the shell command ls, and we can see the utils. py file has been written and that the content is what we had in the cell. As I mentioned in the slides, magic commands also work with the Jupyter Notebook help system. Here we can see that writefile accepts an option -a, which will append to the specified file as opposed to creating a new file. The history magic command has many more options and arguments. I can just print all of the history by default, and remember that this is limited to the current notebook session. Using the -n option, the magic will add line numbers for reference. Notice that only the contents of the cells and not the output is displayed, just like the history command in the shell. However, adding the o option will print the output as well. The l option will only show the specified number of recent entries, and the f option will write to a file. The load and run magics are used to access code in external Python files. I have on my system two very simple Python files, load_demo and run_demo. Both of them simply define a string, and then a function to print that string, and then call the function. The only difference is the text of the message displayed. The load magic will load the contents of a file into the current cell, but it will not run it. The contents of load_demo. py has been placed in the cell; however, there is no output as the print_message function call is ignored because the cell was not run. Also, I cannot access the members of the file such as message. This will give me a NameError. Now I could execute the cell, however, to prevent this. The run magic will also load the code, but instead of displaying it in the cell it will execute the code and display the output. And now I will have access to the members in the file such as message. The alias magic works similar to aliases defined in a shell profile, such as. Bash_profile. The difference again is that these are specific to the Jupyter Notebook. It's useful because you can treat the notebook as a sandbox. One thing to note here is naming conflicts. When matched up against magic commands and Python functions with the same name, the alias has lowest precedence. Here's a simple alias. When run, this alias will simply display a long listing of all of the files in the current directory with a. py extension. Notice that in the alias, I do not have to prefix the ls command with a bang, or exclamation point. Now to execute the alias, I prefix it with a percent sign. Aliases can also accept input with the %s specifier. Now I can pass a value to this alias and it will be used as the extension on which to filter. And finally, if I've experimented to the point of confusion, the reset magic will restore all of the defaults. Note that I have to confirm, and after that all of the aliases are removed and all of the loaded variables and functions as well. Again, this is just a sampling of the available magic commands. There are many more listed in the documentation, and remember to stay tuned for later in the course when we'll create a custom magic command.

## Demo: Security Concerns

At the risk of ending the module on a bad note, I do want to show how potential security concerns could affect the system running a notebook server. For the purposes of this demo, I've set up a Jupyter Notebook server on a virtual machine in the cloud. To make it visible, I've passed the ip option to the notebook with the value 0. 0. 0. 0 to listen on all IPs. I've left the default port as 8888 and just configured my firewall to allow traffic on that port. Now I can access the server by copying this address with the security token. Again, it is possible to change or even disable the security token, but I want to show that this token only permits access to the server itself and has nothing to do with permissions. Notice that I'm connecting to the server using incognito mode in Chrome, so I am not logged in as a particular user that might have saved credentials. This is as anonymous as I can get. So, I'll create a notebook. Right away I can begin to wander around the server. I can also use Python to poke around. The only reassuring thought here is that the permissions are limited to the user running the notebook server, so if I try to access the root directory it won't allow that. But we shouldn't be running as root anyway. Now I'm going to go back to the notebook home page. Under New, I'll click Terminal to create a new terminal running inside of the browser. Now it's as if I'm logged in via SSH. I still can't access the root directory, but I can access any applications that are running such as a MongoDB database. But even if users are not malicious, it is possible to inadvertently create problems with an open notebook server. Here I have a folder for two users. Each folder has an exercise notebook for a workshop, and there is also a shared data folder. I'll connect in this browser as user_1 and connect in another browser in anonymous mode as user_2. Notice that I can work as user_1 and it does not affect user_2's notebook; however, the data folder is still shared, so user_2 will see changes made by user_1, and vice versa. Also, I can go back to user_1's notebook and navigate to user_2's directory. This is because by default Jupyter Notebook is not meant to be multiuser. However, there is the JupyterHub project. This is a multiuser server that is designed for the type of situation I just displayed. Again, this is a topic outside of the scope of the course, but you can learn more about JupyterHub at jupyterhub. readthedocs. io.

# Displaying Rich Content

## Using Markdown

This module we'll take a look at rich data formats that we can use in Jupyter Notebook. Towards the end of module 2 we saw how Jupyter Notebook will render pandas data frames as HTML tables. We also saw that the presentation of those tables can be customized and controlled with CSS rules. In this module, we will see several other ways to present data that are more expressive than just text and code. Comments are useful for describing what is going on inside of code, and as you might expect, you can add comments in Jupyter Notebook code cells. Jupyter Notebook will format them just like in the modern text editor. However, comments are generally best used when describing the code itself as opposed to narrating the justification for writing that particular code. Also, the comments are limited to plain text, so emphasizing certain words or using a list will be cumbersome. We need to be able to render rich text with different type styles, bullets, and more, and we can do this with something called Markdown. If you've used GitHub, you've likely come across Markdown when writing README files or comments. Cells in Jupyter Notebook can contain Markdown syntax that is very similar to that of GitHub. When a cell that contains Markdown syntax is run, Jupyter will render the Markdown in the cell. So let's take a look. The stars around the word very are part of Markdown syntax; it tells the notebook to render the text enclosed as bold; however, if I try to run the cell right now, I'll get an error. Jupyter Notebook still thinks the cell contains Python code and the content is not valid Python. So I need to change the type of the cell, which I can do in command mode. I'm already in command mode, so with the cell highlighted I can press the M key to change the type of the cell to Markdown. Notice that several things happened. First, this drop-down box now has Markdown instead of Code; Code is the default. Second, the in and out indexes to the left of the cell have disappeared. The content of Markdown cells is not stored in the in and out objects. And the Python highlighting for the cell is gone and the cell is highlighting the Markdown syntax. Notice that the text is in the stars and the stars is bold. This time if I execute the cell, we see rich formatted text rendered as expected in place of the cell. Now back in module 2, we saw how to insert HTML into the notebook, but this is much simpler and faster. So even though it's a new syntax, it's worth the trouble to learn it. And Markdown is used in many other places besides Jupyter Notebook. Be sure to remember that the M key in command mode will change the type of the cell to Markdown. Now it is very possible and likely to make a typo at some time and press the M key while a cell with code is highlighted. Now this would actually render as valid Markdown, the single hash is a syntax for a large bold header, but it wouldn't do our code any good. In this case, simply press Y, again in command mode, to change the cell type back to Code. And you can get very fancy, for example, create a table using pipes to separate the columns and the hyphens to separate the headers. And you can even style the Markdown further within the table. One thing to note. A cell cannot contain both Markdown and executable code, it must be one or the other. If you need to put code inside of Markdown, you can use GitHub style triple backticks. Support for languages other than Python is also included; however, this code is rendered statically, it is not executable. You can check out the reference for Markdown syntax on the Daring Fireball website at daringfireball. net.

## Including Visualizations

I've mentioned several times that one of the biggest applications of Jupyter Notebook is data science, and data scientists like visualizations. Jupyter Notebook lets you easily create visualizations right inside of the notebook. We'll use the library matplotlib, which is part of the Python data stack, to do this. The module for the API has a long name, so when I import it I'll alias it as well. Now to get the visualizations to show up in the notebook. For this, I'll use another magic command. This magic command tells the visualization to be rendered inline with the cells instead of opening in a new window. Now I need some data to display. I'm just going to generate some random data using the NumPy numerical computing package, so I'll need to import it as well. Next I'm going to select items from a normal random distribution. To review, a normal distribution is one in which the values tend towards the mean of the range, by default the mean is 0. But before I do that, I'm going to set the random seed to a specific number. NumPy will generate the same set of numbers every time for a specific seed. This way if you're following along, you will see exactly what I am getting in the video. Also, some runs work better than others, and this way we can get reproducible data. Now I'll create the data. So here I've selected 100 different values from a normal random distribution with a mean of 0. Now when displayed in a histogram, the chart should have a single peak in the center, and this is also known as a bell curve. There is one single peak in the graph like expected, but it's kind of hard to see, and we'll look into this more later. But now, let's try a different distribution, the uniform distribution. In this distribution, all of the values in the range have an equal probability of being selected, so the graph should be relatively flat. Hmm, this is even less obvious than the normal distribution, so a data scientist would begin to experiment, maybe a different type of chart would help to see what is the problem. So how about a scatter plot? This will be a two-dimensional graph, so I'll need a set of data for the Y axis as well. Plotting the normal distribution, we would expect to see the values towards the center of the graph. Now this isn't really that obvious for a couple of reasons. First of all, note that the center of the range is not in the center of the graph because the outlier is on the other side. But it is still hard to tell if the normal distribution is working exactly the way we expect it to. So, how about the uniform distribution? A scatter plot should evenly cover the graph in this case, and this isn't a whole lot better. The problem is that we need more data. In other words, our sample size is too small. This is where the power of Jupyter Notebook comes in really handy. Since the sample size is in the size constant, by experimenting with different values of size for the sample, we can see if our predictions were correct. So I'll change size to 10, 000. Now remember back to the previous module. I don't need to hit Shift+Enter a bunch of times unless I want to see each step as it is run. Instead, I can use the Run All Below option in the Cell menu, which again, will also run selected cell. Now looking at the normal distribution histogram, there is a definite peak at about 0. The uniform histogram is relatively level. The normal scatter plot is most crowded in the center, and this time the center of the range and the graph are the same because the larger sample size distributed the outliers as well, and the uniform scatter plot covers the entire graph. And if I wanted to make things a little bit more explicit with the normal histogram, I can set the number of bins in the graph to 50. And now the shape of the bell curve is much more obvious. This is just a snapshot of the capabilities of matplotlib, a full discussion is well beyond the scope of this course. Check out the documentation for matplotlib at matplotlib. org. While browsing the documentation, you may notice something about matplotlib, it's very extensive, with two stars around extensive. You can do anything you want with it, but it might take a lot of code to get there. Fortunately, there's another library called seaborn, which is part of the PyData project, which includes a number of popular visualizations. The nice thing about seaborn is that it leverages matplotlib, so anything displayed with seaborn can be used in Jupyter Notebook as well. For a brief demo, I'll go to the seaborn home page at seaborn. pydata. org. I'll go to the example gallery, and I'll click on this first example. Then I'll copy the source. Back in Jupyter Notebook, I'll paste it in a new cell and then run it with Shift+Enter. I don't need to do anything else in this notebook because I used the matplotlib magic command previously, but in the new notebook you would have to include it before rendering seaborn graphs.

## Applications to Data Analysis

Let's get back to our weather data. There are number of useful ways to visualize this. For example, we might want to see which month in Nashville is the wettest. So at first I'll use pandas to read the csv file with the weather data into a data frame. Now to get the row for Nashville rain, I'll use loc on the data frame with a value of 2, as this is 0 indexed. This returns another pandas data type called a series, which is conceptually similar to a Python dictionary. But the important thing I want to emphasize here is the data type for the values in the series, it's object, and that's because the city and precip are strings and everything else is a floating point. Pandas doesn't know how to resolve these two types, so it bails and assumes everything to be the uber type object. But we already know that we are looking at rainfall from Nashville, so we don't need the first two items, thus I'll remove them with a slice. The resulting series will still be of type object. Pandas won't change the data types for the new series; however, using the astype method will allow me to cast the values in the series to a different type, float 32 in this case, and that comes from NumPy. Now to find the wettest month is not difficult using pandas, but what if we want to know the minimum? That's easy as well. However, this is beginning to be a lot of typing. Also, code is only useful for people who understand it, but just about everyone can look at a bar chart and see instantly the maximum and minimum values. And this is why pandas has methods for visualizing data frames in series. So here's how to create a bar chart. Now we can see at an instant the wettest month is May and the driest month is October, though August is in a close second. Let's try another one. How about which city is the snowiest. First, I'll need to get only the rows in the data frame for which the precip column is snow, and once again pandas makes this easy. Now I need to remove the city and precip columns. Pandas will let you select columns in a data frame using a slice. Don't get too worried about this syntax right now because it's beyond the scope of this course, although we will see it again in the next module. Check out the pandas documentation at pandas. pydata. org for more information. Now I want to get to the totals, and there's a method for that, but this sums the columns, and I want to sum the rows to get the totals for each city. I can do this by passing an axis to the sum method. By default, the axis is set to 0, which is columns. As you might guess, rows is 1. And now I can plot this. I'll use a pie chart to start off with. On second thought, maybe a bar chart is the right choice, but instead of using a vertical one, I'll use a horizontal one. And I can easily add the city names back in, and now it's obvious that Knoxville gets the most snow in the state of Tennessee. I know I have labored a lot over visualizations in Jupyter Notebook, and that might seem like overkill because truthfully everything in this video could be accomplished without Jupyter Notebook; however, notice that the process of seeing the intermediate steps and staying in one application made getting the final outcome much easier.

### Displaying Images

There's one more type of content which might be useful in a Jupyter Notebook, and that is images. Jupyter Notebook can display images stored locally or from the internet. So first, I'm going to need an image. It's easy to create images from matplotlib visualizations, so I'll create one from the snow bar chart from the previous video. Now I can see the new file. To display this image, I'll import the image class from the IPython. display module. The image was clipped a little bit, but that's a matplotlib issue that's beyond the scope of this course. Also, we can use Markdown to display images. Images can also be shown from an URL. I'll use this placeholder image from the site Picsum. To show this in Jupyter Notebook, I'll assign the URL to the URL keyword argument in the image initializer.

# Extending the Notebook User Interface

### Introducing Widgets

This module will explore features of Jupyter Notebook that will take your experience and productivity to the new level. First, I'll show off widgets, which bring Jupyter Notebook to life with interactive JavaScript controls embedded inside of the notebook. Then we'll look at making custom magic commands to further extend the power of Jupyter Notebook. So far, in order to modify the output of cells in Jupyter Notebook, we've had to write code. For example, in the previous module we saw the result of graphing two different types of random distribution; however, it was only obvious from the graphs that those distributions worked as advertised after the sample sizes were sufficiently large. In order to change the sample size, we had to modify the value of the size constant in code, and then rerun the cells. Now this was a simple example, but there are larger cases in which modifying the code directly could be cumbersome and error prone, and for nontechnical people, it might be out of the question. Jupyter supports the embedding of interactive JavaScript widgets inside of a notebook. This enables a more user-friendly interface to our Python experiments. Instead of relying upon modifying the value of the sample size in code, we could provide a text box instead. Let's see how to create these widgets. The base set of Jupyter Notebook widgets is found in the ipywidgets package. This is not part of the default Jupyter Notebook installation. It can be installed with the package manager for your Python distribution. I'm using Anaconda, so I'll run this command. Notice that I have to install it using the conda-forge channel. If you're using the python. org distribution, you can simply use pip with this command. Now installing with pip is going to take an extra step of explicitly enabling the Jupyter Notebook extension, whereas Anaconda does this for you. To enable the ipywidgets extension, use this command. Either of these methods will enable you to import ipywidgets into a Jupyter Notebook and create the interactive widgets. So next, I'll fire up the server and open a new notebook. From the ipywidgets module, I'll bring in the interact function. The interact function is going to take a function, which will compute the return value of a widget. For now, I'm just going to use an identity function, which will simply echo the function's input. Next, I'll pass ident as the first parameter to react, followed by keyword arguments for each of the parameters of ident. The interact function will render a widget based on the type of the val keyword argument. For numerical values, it will display a slider. Notice that the slider is labeled with the name of the parameter to the ident function. Also, when I move the slider around, the value of the slider and the value returned by ident are mirrored as a consequence of using the identity function. To make this more clear, I'll have this function return the input +1. Of course this is no longer an identity function, but notice what happens now. The value of the slider is always one less than the output, but I'm letting Jupyter infer a lot here. By passing a tuple of values to the keyword argument, I can specify a minimum value, a maximum value, and a step value. This will render a slider with a minimum of 0, a maximum of 1000, and the slider will move in steps of 100. I could also have a function with more than one parameter. Passing add to interact will require two keyword arguments. This results in two sliders, one for each numeric keyword argument, and as I move the sliders around, notice that the result is the sum of the values in the sliders. However, interfaces are more diverse than sliders and data types are more diverse than numbers. So I'll explore what other control type mappings there are. First, I'm going to return the identity function back to the first implementation. Now I'll call interact with the identity function again, this time with a string for the keyword argument. This time the control is a text box, which is a logical choice, and if I change the value of the text box, the return value changes as well. Now I can also write a function like this, and I can use that with the interact function. Now the return value is the length of the content in the text box, and if I change it, the return value of the function will update. Passing a Boolean value will result in a checkbox. And I can create a drop-down box by passing a list to the keyword argument. The value of a drop-down will reflect the value of the selected item; however, you can also pass a dictionary to have the value of the drop-down be a different yet associated value. It turns out that there are 37 different widgets in the ipywidgets package, letting Jupyter Notebook infer which is the best one is not always appropriate, and sometimes it's not even possible. Each of the widgets has a specific type in the ipywidgets package. For example, the slider is of type intSlider. If the initial value were floating point, it would be a float slider. But there's also a range slider that has two handles, and the value of the range slider is the range in between them. To create one of these, it must be explicitly instantiated. And there are even more complex widgets such as a color picker. Clicking on this widget will display the system color picker dialog. I'm on a Mac, so on Windows or Linux you'll see something different, but the value of the widget will be the hexadecimal number representation of the selected color. It's also worth mentioning that you can apply interact using a decorator. There's nothing really different about this method except that it is syntactic sugar; the results are still the same. There are also layout widgets and even a widget for accepting input from a game controller. Throughout the rest of this module, I'll demonstrate how to use more widgets and how to use them together to create interactive dashboards right inside of Jupyter Notebook.

## Using Maps with Widgets

There are also third-party widgets which you can install. For example, the ipyleaflet project has a widget that will embed an interactive map in a Jupyter Notebook, so let's try it out. At the command line, I'll install the ipyleaflet package. Like ipywidgets, this package can also be installed from pip, and the extension will need to be enabled as well. See the ipyleaflet documentation for more info. After that is complete, I can start the server back up and create a new notebook. Next, I'll import the map widget from the ipyleaflet module. Now I'm going to create a map and pass it two keyword arguments. The first is a list with the latitude and longitude for the center of the map, and the second is a zoom level. I'll use the coordinates for Nashville, the capital city of Tennessee, for the center, and then I'll zoom in to level 10. Now I can simply display the map. The widget that is displayed contains a map centered on Nashville, Tennessee. I can drag the map around, and I can also zoom in and out. I can also control these features by setting properties on the map widget itself. So if I wanted to zoom out, I could set the zoom level, and it will zoom out much further. We'll see more uses of ipyleaflet in later videos.

## Handling Widget Events

The interact function is just one way to work with Jupyter Notebook widgets. Sometimes, the type of control is not obvious from the type of the corresponding value, and some controls are too complex to be inferred, thus Jupyter Notebook allows us to have more control over the configuration and display of widgets. To enable this, I'll first import the entire ipywidgets module and alias it as widgets. This is a convention you'll see repeated in many widget projects. The widgets module includes a drop-down widget as well. Like is common in user interfaces, a drop-down widget allows a user to select one of a group of items and the widget displays the value of the selected item. Here's how to create one. This code will construct a drop-down widget with the items in the options keyword argument. It will also set the selected item to the value keyword argument. The value keyword argument is optional. If it is omitted, the first option will be selected. To display this, I'll need the display function from the IPython. display module. Now I can pass the widget to the display function. I can change the value that is selected, I can also get that value, and I can set that value. Widgets can also react to events, for example, there's a button widget. The description keyword argument will be the text displayed on the button, but the button by itself is pretty useless. Normally, we expect a button to perform some action when it is clicked. The on_click method of the button will take a function, which will be invoked when the button is clicked. So I'll go ahead and write that function. The parameter to the btn_clicked function is the button that was clicked. When invoked, this function will simply display the current time. To register it with the button widget, pass this function to the on_click method. Now every time the button is clicked, the btn_clicked function will be invoked, which will print the current time. Notice that the output is in the same cell as the button. Let's revisit the drop-down widget. The drop-down widget exposes the observe method, which takes a function that will be invoked when the value of the drop-down changes. The function will be passed a value called a bunch with information about the changes that took place. So first I'll write the observer function, and then I'll register it with the city drop-down widget using the observe method. The keyword argument to observe will filter the type of bunch. Here I just want information about the value that changed. Now I'll select a new value in the drop-down. The bunch is represented as a dictionary, and it has information about the change. Mostly what I'm interested in about here is the new key, which is the value that was just selected.

## Controlling Widget Output

Now we have enough knowledge that we can build a simple interface to go along with our weather data and do ad hoc queries. So first, I'll need to get the weather data into a data frame again. Suppose we'd like to filter the data frame by city and/or precipitation type. This is enough to do with pandas, for example, to see all data for the city of Memphis. But like we saw earlier in the course, while pandas makes it easier to do things in code, doing things in code is not always the best solution. Here, a drop-down box would be much more appropriate. First, I'm going to need to get a list of the unique values for each column that I want to filter by. The unique method returns a NumPy array, but I'm casting it to a Python list. The reason for this will be clear later on in the course. The NumPy array will work just as well though. And as you can see, the list of cities has four values that are all different. Now setting up the drop-down box is just like what we saw in the previous video. Next, I'll create a function for the observer handler that will filter the data frame by the value selected in the drop-down. I'll register the observer with the city drop-down and make sure that observe sends a bunch regarding the values. Now I'll display the drop-down again, and as I change the value in the drop-down box, the data frame will be filtered by the city. However, the output of the drop-down is accumulating in this cell, and I would like for it to refresh each time. The solution to this is to capture the cell output in a special type of widget, and then display the output of that widget in another cell, then we can independently control the output widget by clearing it before displaying a new query. So here's how it works. There's a widget called output in the ipywidgets module that needs to be imported. I also need to create a new instance of output. I'll create it in this first cell because I'll be using it in several places throughout the rest of the notebook. Now in the city observer, I'm going to tell the output widget to capture the output of the data frame by calling display inside of a with block. In the cell after the drop-down, I'll display the output widget. Now the output of the drop-down will be displayed in the new cell. To clear this output widget each time the drop-down changes, I'll add a call to the clear_output method of the output widget in the observe handler. Let me review what's happening again. Instead of rendering the output of the observe method in the same cell as the drop-down, I'm sending it to this output widget and then displaying the output widget in a new cell, that way I can independently clear the output widget without clearing the drop-down widget as well. It might seem a little circuitous, but it's the way Jupyter Notebook works. To ensure that everything works properly, this time I'll select the import cell, and then in the menu Cell, Run All Below. Also, I don't need to be displaying this drop-down here, so I'll delete that line. Now if I change the value of the drop-down, it refreshes the cell below without showing the previous queries as well. Now it should be easy to add another drop-down, which will filter by precipitation type. But there's a problem here. The query only filters by the drop-down, which was recently changed. I want it to work on both values together. So what I really need is a new function to do the querying and have the observe handlers call it, passing the values for the query parameters. The dd_city observer method will call this new method, query_weather, passing the new value from the bunch, but also the current value of the dd_precip drop-down, and the dd_precip observer method will call query_weather as well, except with the value of the dd_city drop-down and then its new value. First, I'll filter on the city. Since there are now two query parameters, I need to take into account the possibility that one is not set. I'll consider that value to be all. If the value of city or precip is all, I won't do any filtering. I'll add the all option to the drop-down widgets later, but I'm storing the filtered data frame, that way I can capture the city and precipitation at the same time. Now I can just clear the output and capture the display of the new data frame. One last thing. To get the initial output, I'll capture the data frame right after it is loaded. Now I'll update my observers and have them call the query_weather function. For safety, I'll run all of the cells from the import cell again. Now I just select a new value from the drop-downs, the data frame updates, except this time considering the values of both of the drop-downs. One last thing I need to do is add the All value into the list of options for each of the drop-down. Now notice that All does not filter on a particular drop-down, but that selecting values will filter on both of the drop-downs. So what we've got here is a small dashboard, and it's an interactive dashboard, and it only uses a single type of control. So what could we build if we used more of the 37 controls in the ipywidgets package? Stay tuned to find out.

## Constructing a Simple Dashboard

Whenever I explain a complex topic like Jupyter Notebook, I try to make all of the concepts fit together, and now is great opportunity to do that. Remember in the second module where I demonstrated how to highlight values in the data frame based on certain criteria? Well again, this was done in code with constant values, but what if we added a text box where a user could type in a custom value? So let's see what happens. First, I'll create a text box widget for the value. Now ipywidgets includes many types of text inputs like the standard text box and text area you're likely familiar with from web UIs. However, these will always store their values as strings, so we would have to parse the text into a number and handle errors, and all that boilerplate code. While you're free to do this, I'd recommend checking out the numeric inputs provided by ipywidgets such as IntText and FloatText. I'll be using FloatText here. I can give the FloatText an initial value. Notice that I specified a float and not a string. That's because a FloatText can only hold a float value, and it will also return float values. That saves us some work of casting the text box value to a floating point. Now if I display the floating text, it looks like a standard text box. But there are two changes. First, when the FloatText has focus, it shows these arrows to change the value, but even more important, it won't accept any characters that are invalid in a floating-point number. While you are watching this, I am trying to type letters and punctuation into the box, and the FloatText won't allow it, so that's more work we don't have to do. However, users sometimes like to try to break things, I should know because I'm one of them, and mine get clever and enter an invalid floating-point value using valid floating-point characters, like this. The decimal and negative sign are used in constructing floating-point values, but their use in this input are invalid. If I hit Enter and try to submit this value, the FloatText will simply reject it and restore the previous value. So it's going to be pretty hard to break this, it may frustrate the aforementioned clever users, but it will save us a lot of work. The IntText widget also works in a similar manner. Next I need to write a function to highlight the values. This was covered earlier in the course, so I won't explain it in detail here. Just notice that the value in the conditional is no longer a literal and instead comes from the FloatText box. Now let's hook this up with the dashboard. The observer for the FloatText box merely calls query_weather with the current values of the drop-downs. Next, I'll register that observer, and finally, in query_weather, instead of displaying the precip filtered data frame call the applymap method to apply the conditional styling. Now I'll run the notebook again just to make sure all the cells are run. Now it looks like nothing changed here, and that's because in order to get query_weather to run, I'm going to have to select the value to filter by, but once I do that you'll notice that every value that is greater than or equal to 3. 2 is highlighted in green. Now if I change to snow, and let's go ahead and get all the cities, none of the values are highlighted because everything is less than 3. 2. However, if I highlight the FloatText box and type in a value, say 1. 1, once the text box loses focus, so I'll hit Tab to do that, that will update the minimum highlight value to 1. 1, and now anything 1. 1 or greater will be highlighted. Now there's one more widget that I want to add. Let's use a color picker to select the color of the highlight. First, I'll create a color picker, then I'll display it in the dashboard, I'll write an observer, and register it. Finally, I'll change the string literal green in query_weather to the value of the color picker. Now I'll rerun the notebook to make sure I pick up all the changes. I'll click the color picker widget. The system color picker dialog appears. Again, this is a component which is not part of Jupyter Notebook and is instead is provided by the OS. I'm running a Mac, and this the macOS color picker, but I'll select a new color like red, and I'll see the highlight color update. I'll select snow for Chattanooga, I'll change the minimum highlight to 0. 5, and since blue represents cold, that will be a better color. Now if you're still with me, congratulations! We've accomplished a lot, but now there is another problem. The UI is not very well organized. In the next video, I'll go over the layout widgets to make things look better.

## Polishing the Dashboard Layout

In this video, I'll wrap up the weather dashboard. The UI is functional, but it's getting a little vague as well. Our widgets have nothing to describe what they do. This is easy to fix. Widget initializers accept a description keyword argument. The value will be displayed next to the widget in the notebook, similar to a label. Next, I'm going to rearrange the widgets a little. Instead of having them stacked horizontally, I want to put them in two rows. Both of the drop-downs will be on the first row, and the FloatText and color picker will be on the second row. To group the widgets like this, ipywidgets has the hbox and vbox widgets. I'll put the drop-downs in an hbox, and the FloatText and color picker in another. Now I'll put both of these in a vbox. The container widgets will display their children when the container is rendered; therefore, I don't need to do anything other display the vbox. That looks much better. Let's add one more bit of polish. Remember ipyleaflet? It would be useful to have a map showing the location currently selected. I'll start off by importing ipyleaflet and getting the map type. Next, I'll create a new map. This is the latitude and longitude for the center of the state of Tennessee, and I've zoomed out to where the whole state basically fills the widget. And here's what that looks like. Now I need to hook the map up to the city drop-down so that when a new city is selected, the map will center and zoom on that city. I'm going to need a list of the coordinates for the cities, which I have in a JSON file, so I'll load that into a Python dictionary using the JSON module from the Python standard library. The dictionary has five keys, one for each of the cities, and then one for all, which is the center of the state. To make the map center and zoom, I just need to modify the center and zoom properties in the observer for the city drop-down. The items in the drop-down and the keys in the dictionary are the same. In a production application I would have a single source, but this is close enough for a demo. By setting the center property, the map will move the new coordinates to the center of the widget. If it's a city, I want to zoom in closer, which is level 10, but if All is selected, I want to zoom out to show the entire state. Let's see how this works. That's good, but it's taking up a lot of real estate. I'm going to use the tab widget in ipywidgets and put the widgets in one tab and the map in another. First, I'll create a new tab. Even though the name of this control is tab, it actually represents a group of tabs. To put the individual tabs in the control, assign a list to the children property. The first child will be the vbox, which has the widgets, the second will be the map, and each of those will be placed in its own tab. Now I can remove the cell displaying the map, and instead of displaying the vbox I'll just display the tab. And finally, I'll add some titles to the tabs. Okay, let's see how all this works now. I can select a city, switch to the map, and see it zoomed in, and actually I can get rid of this map now. So I can select a new city, I can switch to the map and see it zoomed it, and come back over here. I can set a new highlight value, change the highlight color, choose a precipitation type, select a different city, see where it is, select all the cities, and the map zooms back out to the entire state. So that's the complete dashboard. Now personally, I think that widgets are one of the coolest features of Jupyter Notebook, and this is just scratching the surface. You could build much more complex UIs and interactivity. If you'd like to see more about the other types of widgets and see how to make custom widgets as well, check out the ipywidgets documentation at ipywidgets. readthedocs. io. In the remainder of this module, we'll see how to make a custom magic command.

## Custom Magic Commands

Remember that there are two types of magic commands, line magics and cell magics. To review, a line magic takes a single line of input, whereas a cell magic works on an entire cell. You can create custom magic commands in Jupyter Notebook, which are line magics, cell magics, or both. These are not difficult to create, but there are a few rules to follow. Magic commands are nothing more than regular old Python functions and classes that have some decorators applied from the IPython. core. magic module. To define a line magic, I'll use the register_line_magic decorator. Now I'll write a Python function, which will take a string and simply return the input as uppercase. By applying the register_line_magic decorator to this function, I can use it as a line magic. Now I can use it just like any other built-in line magics. I could also write a cell magic in the same way. For this I'll need the register_cell_magic decorator. Now I'll write a function that will count the number of words in a cell. This function just uses the re module from the Python standard library to write regular expressions to remove all of the non-alphanumeric characters from the cell, and then remove multiple whitespace characters. Then it just splits on whitespace to get the number of words. Notice that this function accepts both a line and a cell. This is because the cell magic could accept options. If so, those would be stored in the line parameter. The content of the cell is in the cell parameter. Now I just need to apply the register_cell_magic decorator, and I can try it out. If you're seeing a pattern here, you might have guessed that there's also a register_line_cell_magic decorator too. I won't write one of these because the only difference between the register_line_cell_magic and the register_cell_magic is that the register_line_cell_magic can be used as either a line magic or a cell magic. In the case of the cell magic, it works no differently than a register_cell_magic decorator. In the case of a line magic, none will be passed to the function as a value of the cell. So that works great, but it's going to be inconvenient to use my new magic commands in a new notebook. There are two ways around this. First, I could put the code for the magic command in a new file, then I can load it with the load_ext_magic command. That's right, there's a magic command to load magic commands. So first, I'm going to copy and paste the count_magic into a file called my_magic. py. A few changes need to be made in order to load this. First, I'm going to remove the decorator; it's not needed for this type of scenario. Next, I'll write a function named load_ipython_extension. Jupyter Notebook will use this function in order to load the magic command. The argument to this function implements the register_magic_function method, which takes the name of a function to register as a magic and a string as to what kind of magic, cell in this case. Now we'll open a new notebook, and in the first cell I'll run the load_ext_magic, and I can use the count_magic cell magic. This method is great for loading specific project magics, but what if you have a really useful custom magic that you want to load into every notebook automatically. Jupyter Notebook will look in the folder for your user profile and execute any code files in it when starting up a new server. On the Mac, the folder is in. ipython/profile_default/startup under the home folder for my user profile. On Windows and Linux, the user profiles may be stored in different places according to the configuration of your system. In this folder is a readme file, which recommends prefixing the names of the files with a numeric value as the files will be loaded in lexicographical order, so in this way you can control the order of execution. I'm going to copy the original version of the magic command, the one with the decorator. Then I'll paste that into a new file called my_magic. py. Actually, I'm going to call it my_global_magic. py and I'm going to prefix it with a 00. In addition, I have to add the import for the register_cell_magic. Now I'm going to open up a terminal and I'm going to move this file into the startup folder. For the file to be loaded and the new magic command to be available, I'll need to restart the Jupyter Notebook server. Now I'll start a new notebook, and I'll be able to use the count_magic command again. The magic command is available for all to use. I also briefly want to mention that there is a class-based method of writing magic commands as well. The magics class in the ipython. core. magics module is the base class for all class-based magic commands. Here's what the class-based version of count_magic would look like. Notice that in addition to inheriting from the magics_class, the magics_class must also apply the magics_class decorator. Unlike function-based magics, to use a class-based magic command, you have to use the load_ipython_extension function seen in the previous example. The main benefits of the class-based magic command are persistent state, as well as access to the complete IPython object as well. These are advanced topics that are beyond the scope of this course. Check out the IPython documentation for more, and yes, I did say IPython. Remember that Jupyter Notebook started off as IPython, so a lot of documentation for IPython applies to the Jupyter project as well. Congratulations! This was a big module with a lot of content, and we've still one more to go. In the next module, we'll see how to use Jupyter Notebook to share and collaborate effectively. After that, you'll be a Jupyter Notebook ninja.

# Collaborating on Notebooks

## Collaborating on GitHub

In this module, we will see features of Jupyter Notebook that make sharing and collaboration much easier. Here I've got a simple Flask application. It's not anything super impressive because this is not a course on Flask. The idea behind it is that you enter a tag in this text box, and then the application will parse the Stack Overflow jobs RSS feed and look for jobs with that tag, so I could look for Python. And there are 162 jobs tagged with Python. For Django, there are 11 jobs, and so on. But that's not really what is important. What I want to point out is that this application has dependencies. If I look in the app. py file at the source, at the top there are two imports for Flask and feedparser. Python has an excellent RSS parsing library that is feedparser, and Flask is the microframework that is running the server. However, Flask and feedparser are not distributed with Python. The collections module that contains the Counter class is part of the Python standard library which is distributed with Python, and Flask even has other dependencies as well. If I were to put this application on GitHub, anyone cloning the repository would have to go hunt down the dependencies of Flask and feedparser and install them manually. For this application it isn't a big deal because it's so small, but a real-world application would be much more difficult. Now I'm going to simulate cloning the application in GitHub. Fortunately, Python lets me store my application dependencies in a file, and then use that file to restore them later. To get this list from the command line, I'll run this command. Now to store it in a file, I'll just redirect the output to a file name. Now the commands I'm using here are for Anaconda on a Mac. If you're using the Python. org distribution or are on Windows, it's going to be different, but don't worry about that because I'm proving a point here that transcends the platforms. Now, what I'm going to do is I'm going to open a new terminal and create a new environment. I'll activate it and list the installed Python modules in this environment. Notice that Flask and feedparser are missing. So I'm going to create a new folder to copy this application, and now I'll copy the contents of the app, and now I'll try to run the application. It fails because it can't find the module with feedparser. So what I'll do is I'll restore the dependencies, and now if I try to run the application, it will work. Now the point of this exercise is to emphasize something called reproducibility. By storing the dependencies in the requirements. txt file, I am able to reproduce the development environment with Anaconda. So how does this apply to Jupyter Notebook? Take a look at this Jupyter Notebook that I have. It works with the weather data that we have seen throughout the course. In addition to code, the notebook contains Markdown, custom-rendered data frames, visualizations, and a widget. Let's say I want to share this on GitHub, so I'm going to go to GitHub and I'll create a new repository. Next, I'll initialize the folder containing the notebook as a Git repository. I'll add the notebook to start tracking it, and then I'll commit the changes. Next, I'll copy and paste the remote from GitHub, and finally I'll push the changes. Now when I refresh the repository and click on the link for the notebook, I'll see that GitHub will try to render the notebook for me. So I can see my code, my Markdown, my data frame, my visualizations, but it does not show the custom highlighting for these data frames, and when I get to the widget it bails completely, and that's because this is a static rendering of the notebook. So let's take a look at the raw notebook data. It's actually a JSON file, and there is an object for each cell, and here's the cell type. This one is code and this one is Markdown. Notice that for the data frame it stores the rendered HTML, but it does not store style information. Down here is a visualization, and notice that it is storing the image data inline; however, that's as far as GitHub is willing to go. That's not dismissing this as useless, for many cases it will do just fine. And don't forget that all the features of GitHub, such as pull requests and bug tracking, will work on notebooks as well, but we can't run this on a live server or get the complete functionality of the notebook. In the next video, we'll see a solution for that.

## Collaborating on Azure Notebooks

In the beginning of the course, I mentioned that Microsoft Azure will host the Jupyter Notebook in the cloud and for free. There are also some sharing features included with Azure Notebooks, and in this video I'll explore some of these. Recall that all you need to use Azure Notebooks is a Microsoft account. I'm using an outlook. com email address, which you can also get for free. I'm inside of this Azure Notebook library, and I'm going to click the New button. This dialog, which I showed at the beginning of the course, has three tabs. This first tab is for creating content from scratch, the last tab I can use to clone content from a URL, including on GitHub. So if I copy the URL for the Jupyter Notebook in the previous video and paste it in the dialog, Azure will clone the contents. Now I can open the cloned notebook in Azure. The main difference between GitHub and Azure is that on Azure the notebook is running on a live server, so I can interact with it, whereas on GitHub it was read only. Now before the notebook can actually run, I'll need to add the data file. This was unneeded on GitHub to render the current state of the notebook, although in the real world you would want to include it for completeness. But in Azure, I can either create a new file and copy and paste the data, which is what I will do here, or I can upload the file from my local machine. So I'll go back to the CSV file, copy the contents, and paste it into Azure. Now I can go ahead and run the cells in the notebook. I still get the code, the Markdown, the visualizations, like on GitHub, but I'll also get the custom styles and the widget also works. Now a word of warning about Azure Notebooks. First, because this is running on a public server, be careful of what you post. Second, this is a free service, and thus the servers are shared among users. This means that you could experience lag, especially when doing installs. I've removed any lags for this course through the magic of editing. Sometimes you might get an error about exceeding a quota for computing resources during a long-running cell. In this case, you might be able to split the cell into several other cells, remember the keyboard shortcut Ctrl+Shift+Minus, and then execute the task in parts to avoid exceeding quotas, but overall, this is a very good resource for learning, and just about everything in the course should be possible in Azure Notebooks. And I can share libraries from within Azure Notebooks as well. So by making this library publicly available with the URL, other Azure Notebook users could clone it and make modifications and try them out live as well, and also, this is free. But that's not all. You can export notebooks into many different formats such as PDF. Azure Notebooks come with an integrated terminal, so you could run conda and pip and install packages and extensions, such as ipyleaflet, and you can even set startup scripts for when notebooks are launched. In addition to ipywidgets, there are several powerful extensions installed in Azure Notebooks. One, which is called RISE, will reformat Jupyter Notebooks and show them as a slideshow, another called bqplot lets you create interactive visualizations without having to use low-level APIs. I encourage you to check out these features by exploring the Azure Notebook documentation, and don't forget that these are not specific to Azure Notebook. You can install these extensions locally on Windows, Mac, and Linux. Azure just packages everything for you to get started easily.

## JupyterLab: The Future of Jupyter Notebook

There is one last thing I want to show you in this course. It's the next step in the evolution of Jupyter Notebook, and it's called JupyterLab. JupyterLab can easily be installed with Anaconda, and there are also Docker images available from Docker Hub. However, Azure Notebooks has it already preinstalled, and that's going to be the easiest way to see it in action. Before showing it, I need to provide a caveat. JupyterLab is not completely baked yet. In fact, it went into beta just before the publication deadline for this course, so what you see in the video may have significant differences compared to the version available when you were watching this course. However, the core concepts should be the same as the project Jupyter announcement about the beta stated that it was "ready for users", but it's so cool that I can't resist showing it. To launch JupyterLab in Azure Notebooks, you need to use a special URL for a library. Note that I said a library and not a notebook, and the reason for that will be clear in a minute. The URL has the format libraryusername. notebooks. azure. com/nb/lab. The last part of the URL, /lab, is the important part that tells Azure Notebooks to open up JupyterLab. Now fortunately, if you have a notebook open, all you need to do is delete this last part of the URL and replace it with lab. Now earlier in the course I emphasized that Jupyter Notebook was not an IDE, but rather an interactive computing environment. JupyterLab takes inspiration from IDEs and adds it to JupyterLab. So over on the left is a tabbed pane, and the one open right now shows all the files in the library. That's why I opened a library in JupyterLab and not a single notebook. In the main part of the UI is this launcher. This is where I can launch a new notebook with several types of kernels from Python to R to F#. I could also open a command-line console with those kernels as well. I could open a terminal or text editor. Now like most IDEs, clicking on a file on the left pane will open it for editing. Now notice that the notebook file that I just opened is in a tab, so I can open up other files in a different tab as well. For example, here is the CSV data. By default it opens in a table view, but if I right-click on the file, I can open it for editing. And JupyterLab takes advantage of pop-up context menus a lot; this is something that Jupyter Notebook ignores. I can rearrange the tabs to have both the data and the notebook visible at the same time, and this is very useful. And then I can create a console window and put it down here, and make changes to the data, and see the results in the notebook at the same time. Now you can also go insane with customization. For example, in advanced settings, you can create and edit keyboard shortcuts to suit your workflow and preferences. So let's see how JupyterLab can make you more productive. First, I'm going to create a new folder and call it jlab_demo. Next, I'm going to create a new Python 3. 6 notebook. I'll right-click on the name in the file pane, click Rename, and rename it to jobs. I'll also go to the File menu and select File, New, Terminal. This terminal will open in a new tab. I'll drag that new tab to the bottom of the screen and JupyterLab will display both the notebook and the terminal at the same time in a split screen. What I'm going to do here is I'm going to reproduce the jobs demo from the beginning of this module where I parsed the RSS feed from Stack Overflow. So I need to get that data, the RSS feed. The file is kind of large and I don't want to keep wasting bandwidth or waiting on it to download, especially while experimenting, so I'm going to use curl to download it and save it to my Azure Notebook library. The files for the library are stored in the library folder. Now I can refresh this pane and see that the jobs. rss file has been added. Now let's explore the feed and feedparser a little. To do this, I'll just work inside of an IPython console. Sometimes it's easier to use the command line to prototype. So I'll go to the File menu and select New, Console. JupyterLab will ask which kernel to use, and I'll chose the Python 3. 6 kernel, which is the same as the notebook. Then I can drag the Console tab to the right of the terminal and keep everything in view. I'll start off by importing the feedparser module. Then I'll parse the jobs that RSS feed. The jobs themselves are located in the entries key, and there are 1000 of them. So let's look at the structure of a job. This tags key has the data I want to collect, so all I need to do is create a counter to tally the number of times each tag appears. Easy enough, right? And we get an error. Now at this point we could explore the data more to find out what's going on, but to save time I'll skip to the end. It turns out that not all jobs have tags, and for jobs that have no tags, the feed omits the tags key for that job, so I need to filter out the jobs that have no tags. Now if I look at the length of tagged jobs, it's 993, so 7 jobs have no tags. Now I can parse through the tag jobs and get the counts. Great! Now this can be transferred to the notebook. Now I want to visualize this, so I'll bring in matplotlib and the inline magic command. Then I'll extract the labels and values that I want to use in the chart. And finally, I'll create a var chart with the data, don't worry to much about the API here. Again, check out the matplotlib documentation for more details. And to make some room, I can adjust the height of this lower section. Now I'm using a lower resolution that is optimized for recording, but likely you'll be working at a much higher resolution yielding more screen real estate, so things won't be so cramped. Now at this point, I can go in any number of directions, but I'll stop here because this is a good overview. JupyterLab is going to be an incredibly powerful addition to the Jupyter project, and the best part is that your existing investment in Jupyter Notebook carries over, and the only piece of software you need to use it is a web browser. So don't hold back, and jump into JupyterLab by installing it yourself or taking advantage of the free Azure Notebooks, you have nothing to lose.

## Course author

Douglas Starnes

Douglas Starnes is a polyglot ninja and tech community influencer in the Memphis area making stuff that works on more than just the web. He is a co-director of the Memphis Python User Group and a...

## Course info

| | |
|---|---|
| Level | Beginner |
| Rating | ★★★★☆ (41) |
| My rating | ★★★★★ |

| | |
|---|---|
| Duration | 2h 14m |
| Released | 15 Mar 2018 |

Share course