Building Your First Python Analytics Solution

by Janani Ravi

Course Overview

Course Overview

Hi. My name is Janani Ravi, and welcome to this course on Building your First Python Analytics Solution. A little about myself. I have a Masters degree in electrical engineering from Stanford and have worked at companies, such as Microsoft, Google, and Flipkart. At Google, I was one of the first engineers working on real time collaborative editing in Google Docs, and I hold four patents for the timeline technologies. I currently work on my own startup, Loonycorn, a studio for high quality video content. Python has exploded in popularity in recent years largely because it makes analyzing and working with data so incredibly simple. Despite its great success as a prototyping tool, Python is still relatively unproven for large enterprise scale development. In this course, you will gain the ability to identify and use the right development and execution environment for your enterprise. First, you'll learn how Jupyter notebooks, despite their immense popularity, are not quite as robust as fully-fledged integrated development environments, or IDEs. Next, you will discover how different execution environments offer alternative ways of configuring Python libraries, and specifically, how the two most popular conda and pip stack up against each other. You will also explore several different development environments including IDLE, PyCharm, Eclipse, and Spyder. Finally, you'll round out your knowledge by running Python on the major cloud environments, including AWS, Microsoft Azure, and the GCP. When you're finished with this

course, you will have the skills and knowledge to identify the correct development and execution environments for Python in your organizational context.

Getting Started with Python for Analytics

Module Overview

Hi, and welcome to this course on building your first Python analytics solution. In this module, we'll see how you can get started with Python for analytics. We'll start this module off with a discussion of how Python as a technology has democratized the field of data analytics. We'll talk of how Python has combined the simplicity of the SQL querying language and the utility of Excel to be a true game changer. In order to get started working with Python, we'll see how we can install the Python distribution on Windows and Mac OS machines. We'll then see how we can run Python code from a terminal window. We'll see how you can install packages on your local machine using the pip package manager. Pip is the most popular package management system used for Python packages. We'll see how we can use pip to install the virtual env package and use virtual env to create virtual environments for your different projects. We'll then move onto a discussion of the different development environments that you can use for Python. We'll see how we can write Python code using code editors, integrated development environments, as well as Jupyter notebooks. We'll get some hands-on practice writing Python code using a code editor, such as sublime text, and using an online editor, repel.it.

Prerequisites and Course Outline

Before we get to the actual course content, let's take a look at some of the prereqs that you need to have to make the most of your learning. This course assumes that you're comfortable programming using basic Python. You don't need to know Python in a lot of detail, but you can write simple scripts. This course also assumes that you have the ability to install tools and packages on your local machine. If you have no programming experience at all in Python, here is a course on Pluralsight that you might want to study first, Python Fundamentals. Here is a quick look at some of the topics that we'll cover in this course. We'll start off by seeing how we can get started with Python for analytics. Here, we'll get set up with Python on our local machine and see

how we can run Python code using simple code editors. In the module after that, we'll see how we can work with Python using Anaconda and Jupyter notebooks. From prototyping, we'll move onto actual development and we'll explore a few Python IDEs for production scale Python. Here, we'll see how we can run and debug Python code using popular IDEs such as IDLE, PyCharm and Eclipse. And finally, we'll round off this course by seeing how we can run Python code on cloud hosted Jupyter notebooks, and we'll work on three different cloud platforms, Microsoft Azure, the Google cloud platform, and Amazon Web Services.

Python for Data Analytics

Data analytics today is no longer an afterthought in a business organization, it's an important team responsible for driving business decisions. So what does an organization need from a data analyst? Here is a code that should be a guiding principle for any analytics team. When the facts change, I change my mind. What do you do, sir? Companies are increasingly turning towards their analytics team in order to get a thoughtful, fact-based point of view. It's important that their decisions be fact-based built with painstakingly collected data. Raw data in the real world is not easy to work with directly. Data has to be set up right, it has to be prepared so you can then extract thoughtful, balanced insights from this data. When you're working with data, your views need to be balanced. You need to take into account pros, as well as cons. And finally, you need to express a point of view to your management team, to your org. This can be in the form of a prediction, or a recommendation, or a call to action. Any data professional has at his or her disposal two sets of statistical tools. Descriptive statistics which help them identify important elements in the dataset influential statistics help explain those elements via relationships with other elements. Descriptive statistics only serve to describe and summarize data. It looks for no explanations. You seek explanations using influential statistics. These statistical tools correspond exactly with the two hats of a data professional. The first and most important job that they have is to find the dots, identify important elements in a dataset because if they don't find the right data, they won't be able to perform the right kind of analysis, they won't be able to connect the dots in the right way. Connecting the dots involves explaining those elements that were found via relationships with other elements. Finding the dots today involves more than just collecting data. Data is more and more plentiful and all organizations collect all kinds of data. However, it's important that you work with data carefully. Real world data can contain missing values, can contain outliers which may be genuine outliers or just incorrectly recorded points. Data preparation, preprocessing, and exploratory data analysis are important precursors to extracting useful insights from data from building models using data. Once your data is ready, you might

want to extract information from data and that is where you connect the dots, and for this, there are a variety of technologies available. You can work with just spreadsheets, whether Excel or online. You can use programming languages to extract insights. You can perform in-memory processing if you don't have very much data. If you have a large amount of data that is big data, you might want to perform distributed processing on a cluster. And you have to query your process data using a query language such as SQL. SQL can be used with relational databases where data size is small or with data warehouses which store data from multiple sources. All of these technologies are viable options, but really data analytics today is a very powerful field because of Python. Python has truly democratized data analysis more than any technology since Microsoft Excel back in the 80s. When you're using technology for analytics, what is it that you need? What are the essential analytical building blocks? Well you need conditional execution. Execute code based on certain conditions that are satisfied. You need interconnected calculations. You need to bring entities together maybe based on a common attribute. You need to be able to perform aggregations and groupings on your data. You need a repeated execution or iteration which allows you to perform the same kind of processing on all of your records. And finally, to work with data at scale, you need a way to reuse your code. You need composition or reuse of logic. Now that know the essential analytical building blocks, let's see what the different technologies have to offer. Spreadsheets are great for complex interconnected calculations. Spreadsheets allow you to join in much data in very powerful ways. Spreadsheets are also great for rapid prototyping for quickly getting something useful up and running. Now let's say you were to choose SQL for data analytics. SQL works very well when you have to iterate over independent rows and perform the same kind of processing for each row, and the SQL syntaxes very simple and very accessible. Each technology has its own strengths. If you to use spreadsheets for analytics, you can perform conditional operation using the if function within cells. It's possible to perform iteration using copy/paste or macros, but both of these options are fragile and almost unmaintainable. And finally, you can't really use spreadsheets for composition or code reuse. What if we were to use SQL querying or SQL databases for analytics. You can perform conditional operations using the if function within queries. You can perform iteration using queries or cursors. SQL is all about iteration, and you can use composition using views and stored procedures. There is some level of query reuse possible in SQL. However, the technology that covers all of the essential analytical building blocks is Python. Programming languages offer full support for all kinds of analytical operations. You can perform conditional execution using if else commands. You can perform iteration using for and while loops, and of course, programming languages are all about functions which allow you to reuse code and logic using composition.

Python Development Environments

In this clip, we'll discuss some of the development environments that you might choose to use when you're writing code in Python. As we mentioned in the last clip, Python has truly democratized data analytics. Python code is intuitive, simple to write, easy to read, easy to understand. Python combines Excel's ease of prototyping with SQL's simple syntax. On the other hand, Python has yet to prove itself as robust and maintainable as Java for big projects. It's getting there though. Python is constantly evolving, and today, it's often used in production grade enterprise systems. So if you're writing code in Python, what choices do you have? Therefore, prototyping, there is nothing like Jupyter notebooks. This is an open source web application that allows interactive development in Python. It also supports other languages such as R and scholar. Running Python on a Jupyter notebook makes Python extremely accessible. Jupyter notebooks are web-based, they are absolutely free for anyone to download and use. There is no cost and there is no installation hassle. Jupyter notebooks are interactive, which make them great for prototyping. They follow the read evaluate trend loop for instant feedback. All of your results, your plots, your visuals are embedded right there within your browser. Indeed, Jupyter notebooks are so popular that all major cloud platforms run some form of hosted cloud-based Jupyter notebooks to support data analytics and visualization. These cloud-based notebooks are also used to build and train deep learning models. So if you're getting started with coding and you're looking to build a prototype, Jupyter notebooks work just fine, but for enterprise-scale development, you still need an IDE, or an integrated development environment. IDEs are typically desktop-based applications. You need to install them on your local machine. These make software development easy. IDEs include a code editor complete with syntax highlighting and maybe even autocomplete. We also include tools to build, execute, and debug your code, and they often contain extensions or plugins to integrate with a source control repository, such as GitHub. If you've written code in Python, especially if you've built prototypes, it's quite likely you've used the Jupyter notebook environment. It's an execution environment and not an IDE. For large projects that span notebooks involve multiple Python files, involve multiple modules, you need to consider an IDE. So what will an IDE have to offer you? You'll get a code editor with syntax highlighting and autocompletion. You don't need to remember all of the functions that go along with your classes, what input arguments a particular function takes in. The IDE will prompt you with all of this. Autocompletion and syntax highlighting might seem like syntactic sugar, but the execution environment that an IDE has to offer with run, build, and debug tools, that's where an IDE is really powerful. IDEs allow you to set breakpoints while debugging step into functions. Examine all of your variables at runtime. IDEs will have stack traces to follow the path of your execution. This is

not something that Jupyter offers. If you're working on a large-scale project in the real world, you'll need to commit your code to a source control repository, such as GitHub. IDEs offer source control integration. If you're looking for a complete integrated development environment for Python, here are a host of IDEs that you can choose from, PyCharm, IDLE, Spyder, Thonny, Eclipse with the PyDev plugin, Visual Studio. Each of these IDEs have their own strengths and weaknesses, and we'll discuss a few of these in more detail in a later module. We'll also work with some of them. If you're looking only for a code editor with syntax highlighting for Python and you don't want all of the bells and whistles associated with an IDE. Here are some popular code editors, sublime text for both Mac, as well as Windows users, Emacs, and Vim popular mostly with Linux and MacOS users, and Visual Studio Code popular with Windows users. For small projects, these code editors work well. They won't give you the full built-in execution support that you'll get with an IDE. If you want to write code in Python and you don't want to install anything on your local machine, well you can use an online integrated development environment. There are online playgrounds, such as repl.it where you can write Python code within your browser with no installation. There is an interactive execution environment which allows you to run your code as you write it. Online environments often have very interesting features. You can debug and list your code, you can use files and third-party packages, they also offer support for hosting and deployment. Once you've built your models, you can host them online.

Python Packages

When you work with Python, whether it's for analytics or for anything else, you won't be writing all of the code yourself. You'll be making use of Python packages and libraries. What makes a Python programming language so amazing is not just a simple intuitive syntax. Python has an incredibly rich set of third-party libraries. These libraries can be for data science, web development, machine learning, working with HTTP requests, you name it. These libraries are made available by a vast community of developers and most of these libraries are absolutely free for you to use. Many of these libraries are open sourced. There are many contributors, they are freely available, and they are hosted in a comprehensive repository called PyPI. The availability of these vast numbers of libraries for almost any use case that you can think of under the sun is perhaps the single biggest driver of popularity of Python. Let's say you're using Python for data science. There is an entire py data stack of libraries available to you and most of them are hosted in PyPI. PyPI stands for the Python package index. Libraries are reusable bits of code written by other developers made available to you. Libraries in Python are often called packages and these packages encapsulate code in files, which are called modules. A library can contain many utilities.

A library is basically a Python package which is made up of many modules. A package in Python is a unit of directories and files that can be easily imported for use in a Python program. When you create a package in Python, you generally set up all of the code in a special directory structure along with some configuration settings. Packages can contain namespaces, modules, which are individual Python script files and also nested packages. All of these are brought together in a single unit, the Python package. Packages are not restricted to certain communities or developers. Anyone can package up their code for use by other developers. If you've written utility tools that you feel others might find useful, well you can create your own package and make it available. You'll set up your code in a specific structure meant for Python packages, and once you've packaged your code, you can publish it for inclusion on PyPI, the Python package index. Once a package becomes part of the Python package index, these can be easily installed for use in Python programs by anyone who wishes to use these packages. And for this, you need to use a package manager. Two common ways to do this is to use the Conda package manager or the pip package manager. Both of these are popular package managers meant to install packages for use on your local machine. There are a few differences between the two though. When you use pip to install packages, you're installing from the PyPI repository directly. When you're using conda, you're installing from the Anaconda repository. Anaconda is a free and open source distribution for the Python and our programming languages specifically meant for the scientific computing community. When you use the pip package manager that comes along with Python, you're only using it to install Python packages. Conda can be used to install packages, as well as to set up other tools on your machine. When you pip install your packages, what you're installing are wheels that is the source distributions of packages. When you use conda install, you install binaries, even for Python packages. Pip is only used to install Python packages or libraries. If you want to install other tools, such as an interpreter, you need an additional package manager or installer. Conda can be used for all purposes, can install any binaries including other language libraries or interpreters. Pip cannot be used directly to create virtual environments. You need to download and install the virtualenv package and then use that. When you're using conda though, you can create isolated environments to manage different versions of Python.

Demo: Windows - Installing Python and Using Pip to Install Packages

In this demo, we'll see how we can install and work with Python on our Windows machine. There are several distributions of Python available, and you'll work with the Anaconda distribution of Python in a later module. In this module, we'll head over to python.org, go to Downloads, and you

can see that the latest version of Python is Python 3.8. You can also see that this website has automatically detected my operating system as a Windows OS. You can click on this button to download Python or simply click on the Downloads tab. This will take you to a new page and you can download Python 3.8 .0 using this yellow button here. This will bring up a file dialog on your local machine prompting you to save the Python installer. Go ahead and hit Save, and in under a minute, this executable will be downloaded and saved to your local machine and once the download is complete, you can right-click and hit Open and this will bring up a little wizard that will walk you through the Python installation. Click on this checkbox here to add Python 3.8 to your part environment variable that will allow you to access Python from within your terminal window. This is the part on my local machine where Python will be downloaded and installed. I'm fine with this. I'm going to go ahead and move on. This will kickstart your Python installation. You might have to wait for a minute or two until all of the packages are downloaded and set up, and once this is done, click, close, and you're set up was successfully and Python is now available for you to use. Click on the Start button of your Windows machine and select the command prompt. This will open up your terminal window or your DOS shell on your Windows machine and that's where we'll work. Run the python --version command to see where the Python is available and what version we are working with. You can see that we are working Python 3.8 .0. For most of the three-point x versions of Python, the pip package manager is installed automatically. Take a look at pip --version and you can see that we are working with pip version 19.2 .3. If you want to use the Python interactive environment from within your terminal window, simply type in Python and that will take you to the Python 3.8 shell. You can now type Python commands within this terminal window. Hit Enter and these commands will be executed. Hello world has been printed to screen. You can also import and use built-in Python modules or packages. Let's import the math module here that is available by default and let's invoke the sqrt function on the math module. Math.sqrt of 4 gives us the result 2. The math module is available by default, but what if we try and import the NumPy model to work with multidimensional arrays. The NumPy package is not available in this default Python distribution. There is no module named NumPy. If you want to work with NumPy, we need to explicitly install it. Type exit command in order to exit out of this Python shell. And once we are back on the terminal prompt, we can install the NumPy module using a pip install. Pip package manager will download and install the NumPy package from the PYPI package index. Hit Enter and a NumPy will be installed on your local machine. Now that you have NumPy available, let's enter the Python shell once again. Type in Python and you'll enter the interactive shell environment for Python 3.8. Now you can import the NumPy package and it'll be available for you to use. Let's check out the NumPy version and you can see that it's 1.17 .3. Once you have the NumPy package installed on your machine, you can exit out of this shell and re-

enter the shell at a later date. The NumPy package will be available for you to use. Let's import the NumPy module once again and alias it as np. You can now use np to create new arrays. Here is our array. Let's multiply this array by 2 and here is the result right here on your screen.

Demo: MacOS - Using Brew to Install Python 3

In this demo, we'll see how we can install Python 3 on our Mac OS machine using the brew package manager. Here I am with the terminal window open on my Mac OS machine. I'm currently in a directory called BuildingYourFirstPythonAnalyticsSolution. This directory is currently empty. You can run an ls command and you'll see that there is nothing in here. Now Mac machines come with a built-in version of Python. Run python --version and you can see that I currently have version 2.7 .10. In addition to Python 2, I want to install Python 3 on my machine, and I'm going to use the brew package manager for Mac OS and Linux. To get this, head over to brew.sh. Installing the brew package manager is extremely straightforward. Simply copy this command over and switch back to your Mac terminal window or Linux terminal window and run this command. Hit Enter and this will download and install the brew package manager. Now that you have brew successfully installed on your machine, let's take a look at the version of brew that we are working with. This is Homebrew version 2.1 .15. I mentioned this earlier, my Mac has a built-in version of Python. Python --version shows me that it's version 2.7 .10. Now you can confirm that we don't have python3 installed. I'm going to run python3 --version and this gives me nothing. The command is not found. We have Python 2 installed. I'm going to use the brew package manager to get python3. Brew install python3 will do the job for me, just wait until all of the packages are downloaded and installed. This installation might take a little under a minute. Once you're done, Python 3 is available for you to use. Run the command python3 --version and you can see that you have Python3 version 3.7 .4. Most Python3.x versions, I think about Python 3.4, come with the pip package manager. Run pip3 --version and you can see that you have pip version 19.2 .3. On my Mac OS though, the default installation of Python, that is Python 2.7, does not come with the pip package manager. Pip --version shows me no such file or directory. Now generally, if you're starting off, you won't be working much with Python 2. And when you install Python 3, you get pip, but if you have to install pip separately, you can use easy install. Just simply invoke the command sudo easy_install pip. Easy install isn't really used much to install pip. It's a slightly outmoded way, but you should know that it's possible to run easy install to get pip on your Mac OS. Now with this install complete, I have the pip package manager for my Python 2.7 version as well. Pip --version shows me I have 19.3 .1, that is the pip version for Python 2.7. The pip command uses the package manager for Python 2.7. Pip3 is what I use to work with Python 3.

Pip3 --version shows me a different version and you can see that it's for Python 3.7. My Mac machine is currently set up with Python 2, which was already installed also Python 3, which I just installed using brew. Now if I just run the Python command, that will take me to the interactive shell for Python 2.7. This is the default Python installed on my Mac. Within this shell, I can write code in Python to print 7/5, this is a command that works in Python 2, and this executes successfully and it gives me 1. This is integer division. Let's print Hello world without the parentheses. This is correct code in Python 2 and this also executes successfully. If you want to exit out of this Python 2 interactive shell, simply type the exit command and you'll be back on your terminal prompt.

Demo: MacOS - Using Pip to Install Packages

In this demo we'll see how we can use pip to install packages on a Mac OS or a Linux machine. Here I am on the terminal window of my Mac. I have Python 2, which is the default version that is installed on my Mac. I also have Python 3. I'm going to run the Python 3 command in order to get into the interactive shell for Python 3. In this shell, the code that you write has to be in the Python 3 syntax. Print Hello world, make sure you specify the parentheses, and the code will execute successfully. Let's perform a simple arithmetic 1+3 multiplied by 2 and this gives us 8. Everything seems to be good so far. Let's go ahead and import the math module, which is available built-in in Python. The import was successful. There was no error here. I'm going to initialize a variable r equal to 5 and you use this r, which is the radius of a circle to calculate the area of a circle. The constant pi is available in the math module that we just imported. Let's print out the area of a circle with radius 5 and you can see it's 78.539. Everything looks good so far. Let's try and import the NumPy module and oops, module not found. The NumPy module is not available as a part of the standard Python distribution that we install. We'll need to explicitly install a NumPy using a pip install. Exit out of the shell and let's head over to our terminal window and install NumPy using pip3 install. This will install the NumPy package for our Python 3 installation, not for the default Python 2. The NumPy version that was installed is 1.17 .3. Let's head back into our python3 interactive shell. Now that we have NumPy installed on our local machine, let's go ahead and import the NumPy module. The import went through successfully. Let's take a look at the NumPy version we are working with. It's 1.17 .3. We've successfully installed NumPy for Python 3 using pip. Let's exit out of the shell and now I'm going to explicitly install a specific version of NumPy. I use the pip3 install command once again, but the NumPy version that I want is 1.16 .5. This install command will automatically get rid of the earlier installation of NumPy that I had and install this current version that I had specified. You can see from the messages here that pip found an

existing version which it uninstalled and installed the version that we had specified. Alright, let's go back to our python3 interactive shell. I'm going to go ahead and import the NumPy module once again. The import is successful. Let's take a look at the version of NumPy we are working with. It's 1.16 .5, the version that we just installed. Once again, let's exit out of this shell. What about the Python 2 installation that we also have on this machine. Let's get into the Python 2 interactive shell and I'm going to import the NumPy module. Now this import was successful, but if you take a look at the NumPy version here, you'll see that it's something different. It so happens that the default version of Python that came with my Mac machine also had NumPy installed and this particular version of NumPy. If you're working on a Mac and you have multiple Python installations, you should know that the packages that you install for each are different, they don't affect the other. This can be a little confusing at first. Let's head back to our Python 3 interactive terminal and let's work with NumPy. I'm going to import NumPy with the alias np and I'm going to create a new array. I just want to make sure that I can work with the package that I just installed. Array creation is successful. I'm going to create one more array. Array_2 here. Here are the contents of array_2 and I'll perform a simple operation. Array_1 + array_2, that gives me array_3. Everything looks good. I've successfully installed packages using pip for Python 3.

Demo: Installing and Working with Virtual Environments

In this demo, we'll see how we can create virtual Python environments to work in. A virtual environment can be thought of as an isolated time box where you install packages. If you have different projects and they require different versions of these packages, you can run each project within a virtual environment, so they are all essentially isolated from one another. Here I am on the terminal window of my Mac machine. This text that we'll follow here are the same steps that you'll follow if you're using a Windows machine. There is one slight difference in one command and I'll mention it when we get there. Virtual environments can be installed on your machine just like you would any other Python package. Pip3 install virtualenv will install virtualenv for the Python3 installation that I have on my machine. Once the package has been installed, you can create a new virtual environment by invoking the virtualenv command. Pythonenv is the name of the virtual environment and -p python3 indicates that the Python3 interpreter should be used within that virtual environment. Once this command runs through, you've successfully created your virtual environment. The pythonenv virtual environment is available as a folder within your current working directory and you can activate your pythonenv virtual environment by running this command .pythonenv /bin/activate. If you're on a Windows machine, that command is a little different. Simply execute the activate.bat file and the pythonenv\Scripts. You can just run this

command on your Windows shell. Once this is done, you are within your virtual environment. Your terminal prompt will be a little different indicating to you what environment you're working in. If you run an ls command here, you'll see that all of the packages and other details and configuration settings, etc. associated with your virtual environment is available within this folder here, pythonenv is the folder. Virtual environments serve as an isolated sandbox for your Python packages. I'm going to run a pip3 install within the pythonenv virtual environment, and this command will install the pandas package and all of its dependencies, including NumPy, within this pythonenv environment. Remember when we had created this virtual environment we had specified that the interpreter to use here should be the Python 3 interpreter. So if you just run Python, you'll get into Python 3. In this isolated sandbox, Python 2 does not exist, that's in the base environment of our local machine. Now within this interactive shell, let's go ahead and set up an import for the pandas module and the import goes through successfully. So we know that pandas is available within our pythonenv virtual environment. Let's exit out of this interactive shell and I'm going to invoke the deactivate function to deactivate this pythonenv virtual environment. We are back in our base or default environment. Our prompt has changed. It doesn't indicate the name of the environment. Back in our base environment, I'm going to invoke the Python 3 interactive shell and see whether pandas is available here in my base environment. Nope, not here. The packages and the versions of packages that you install within a virtual environment are not available within other environments, thus, you have multiple sandboxes that you can use for your different projects.

Demo: Editing a Python Script Using Nano and Vim

In this demo, we'll see how we can work with two simple code editors to write Python scripts. We'll work with the nano editor and the vi editor, both of which are popular on Linux and Mac machines. Here I am on the terminal window of my Mac machine and I'm going to create a new file using the touch command. Touch file.py will create a new empty file within my current working directory. Ls shows me that file.py is available here. Alright, let's open this file up using the nano editor which is available built-in on the Mac. Nano is a very simple text-based editor which is great for editing Python files. It can also be installed and used on Windows, but on Windows you might just use Notepad. This Python script has just one line of code, print Welcome to Loonycorn. Go ahead and hit Ctrl+X to save and exit the nano editor. This will take you back to your terminal prompt. Hit Yes to save your modified buffer. Once the contents of this file have been saved, back at your terminal prompt, you can execute this file. Simply run python file.py. This will execute your script and print out to screen Welcome to Loonycorn. Now if you just use the

Python command you're invoking Python 2 by default. If you want to run this script using the Python 3 interpreter, you'll have to say Python 3 file.py. If you're working on a Linux machine or on a Mac machine, vim is an extremely popular text editor. Let's see the vim version that we have. It's vim version 8.0. Vim is an extremely popular text editor on a Mac or on a Linux machine. You can install vim on Windows as well if you want to. Vim comes built-in on a Mac. If it's not available, you can get it with a simple brew install. Vim makes heavy use of keyboard shortcuts and it's rather hard for beginners to use, though experts are very quick with the vim. Let's open up file.py using the vim editor. Here is the single print command within this file.py. Now by default, vim opens up in command mode, so you can execute keyboard shortcuts to run commands, but you can't actually edit your file. If you want to execute your Python code, you can type this command, :! python %. This will run the Python interpreter on this file and observe Welcome to Loonycorn is printed to screen. And if you hit Enter to continue, that will take you back to your vim editor. If you want to quit out of vim, you'll hit :q!. Let's open up our file.py script using the vim editor once again. Like I said, the default mode is command mode. In order to get into insert mode, you need to hit the i shortcut. This will take you to insert mode and you can now type and edit this file. I'm going to add in another print statement saying this is a new line. If you want to get out of insert mode and back into command mode, you'll need to hit Esc. Once you hit the Esc key, you can run your keyboard shortcuts and :wq will save and quit your file. Let's open up our vim editor once again, vim file.py. This editor will open up in command mode and I'm going to run :! python % to execute this Python script. Observe that This is a new line is also printed. Hit Enter and we go back into vim, and :q! will quit the file. Vim is kind of hard to use at the beginning, but experienced programmers prefer vim because of how easy the shortcuts make all of the actions.

Demo: Editing a Python Script Using SublimeText

In this demo, we'll download, install, and use the very popular sublime text editor and use it to edit a simple Python script file. This Python script will read in a simple csv file into a pandas data frame. The csv file that we'll use is this advertising.csv. Here is what the file looks like. It contains just a few columns, a row id, TV, radio, and newspaper spends, and the sales of a particular organization. In order to download and install sublime text on our local machine, we'll head over to sublimetext.com /3, that is the version of sublime text that we are going to use. You can install sublime text for OS X, you can also install it for Windows or Linux. since I'm working on a Mac, I'm going to select the OS X option and the dmg file will be downloaded to my machine. I'm going to right-click, go to Show and Finder, and here is the dmg file that I'm going to double-click and install. And here is the Sublime Text icon showing me that it's now available on my machine.

Double-click and open up a new editor. I need to confirm that I indeed want to open this editor that I've downloaded from the internet. This is the default sublime text editor, which I'm now going to switch over to full screen mode. I'll just paste in the contents of my python script file. Observe that we have no color highlighting for our Python. This is because we haven't saved this file as a .py file. I'm going to call this data_exploration.py, and once you save this file, observe that our code is now nicely highlighted. Here on top are the import statements for pandas, as well as NumPy. I'm reading in advertising.csv, my current working directory, into a pandas data frame. I'll print out the first five records in this data frame. I print data.head. I'm going to print out the shape of the dataset, and then going to invoke the describe function on the pandas data frame and print out a quick statistical summary of my data. I'm going to invoke the isnull sum function on my data frame to check to see whether there are any null values and I'm going to print out the data types of the different columns. Make sure you hit Ctrl+S or Cmd+S to save this file and switch over to a terminal window. Let's run an ls command and this file should be present in your current working directory along with the advertising.csv file. If you want to run using python3, you'll need to invoke pip3 install pandas. We've installed NumPy, but not the pandas library. Once you have successfully installed pandas, you can run this Python script. Now use Python if you want to run Python 2, and if you want to run using Python 3, simply use Python 3 data_exploration.py. I'm using Python 2 here, but it's really up to you. You can use Python 3 if you want. I already have pandas installed for Python 2. And you can see that the script runs through successfully and all of what we printed out to screen is displayed within our terminal window.

Demo: Using Online Editors to Write Python Code

If you don't want to install Python on your local machine, but you still want some practice using Python, well you can use an online editor and that's exactly what we'll do here in this demo. We'll work with repl.it. Head over to repl.it on your browser window. This is an online coding platform and it's not just for Python, but it also supports a number of other programming languages. REPL here stands for read evaluate print loop, and it's just a short form for an interactive environment. New REPL shows you all of the languages that are supported here at repl.it. We'll go with Python because that's what we want to work with. Select the Python option here in this drop-down and click on the button that says Create repl. A new editor will open up and you can write code in Python right here within this editor. You can practice your Python code within your browser, no installation required. Off to the right here is the terminal window where your code will be executed and output results will be printed to screen. Off to the left here, you'll see all of the files that you have opened up within this session. Observe that we are not signed in. You can play

around with REPL without signing in, but if you want to work with all that REPL has to offer collaborative team environments and so on, you'll need to sign in. I'm going to use this online editor to write some simple Python code. I initialize a variable x and another variable y, both of these are floating point variables, and I'm going to sum the two and place the result in the variable sum, and I'll print some out to screen. Now if you want to execute this code, simply click this run button here on top. Your code will be executed and your result will be printed out to the terminal on the right. Let's write a little more Python code. I'm going to accept an input from the user, convert that to an integer, and assign it to the variable a. I'm going to calculate a raised to the power 2 and print out a and the square of a. With our code written, all I need to do to run this is hit this Run button here on top. My code will be executed and here on the right I see Enter a number. I enter the number 15, the square of 15 is 225. Let's now use this REPL environment to execute the code that we worked with earlier where we read in a csv file into a pandas data frame and printed out a bunch of information. Observe here on top we have a few import statements. We are using Python packages that are not available with the default Python installation. The rest of the code here is the same code that we saw when we used the sublime text text editor. In order to tell REPL that we need additional packages to be installed, we need to create a new file. This is the requirements.txt file. Create a new file within your session, and within requirements.txt, you need to specify the packages or libraries that you're planning to use, NumPy and pandas. In addition to these modules or packages, our script also needs the advertising.csv file which I'll now upload from my local machine. Select advertising.csv, hit Open, and this will be uploaded to your REPL session. We now have everything that we need for this script to run. Click on the Run button after switching over to main.py and your script will be executed. Observe that the packages will be installed automatically based on what you specified in the requirements.txt file. It seems like the REPL environment already had NumPy, but if you scroll down below, pandas was freshly installed, and this right pane here is basically your terminal. The output of your script is printed out within this terminal window.

Module Summary

And with this demo on writing Python code using the repl.it on line editor, we come to the very end of this module on Getting Started with Python for Data Analytics. We started this module off with a discussion of how the Python programming language has democratized data analysis more than any technology since Microsoft Excel. We then saw how we could install Python 3 on our local Windows machine, as well as a Mac OS machine. Once we had Python successfully installed, we saw how we could use the Python interactive environment from within our terminal

window. We wrote some simple Python code and we also imported a few built-in libraries available in Python, but for additional libraries, we needed to use the pip install to install Python packages. The pip package manager came along with our Python distribution. This is true for most versions in Python 3. We used pip to install additional Python packages, such as NumPy and pandas onto our local machine. We also saw how we could use pip to install the virtual env package. We then saw how we could use virtual env to create isolated Python environments to run different projects. We then moved onto a discussion of the different development environments that you could use to write Python code. We discussed Jupyter notebooks, integrated development environments, and code editors. And finally, we got some hands-on practice writing Python code using simple code editors and online editors. In the next module, we'll see how we can write Python code on Jupyter notebooks.

Working with Python Using Anaconda

Module Overview

Hi, and welcome to this module on working with Python using Anaconda. The Anaconda distribution of Python is one of the most popular distributions for data size. Anaconda also supports the R programming language, but since we are working with Python, here in this module, we'll get a good overview of all that anaconda has to offer for Python programming. The Anaconda distribution of Python comes along with Jupyter notebooks. Jupyter notebooks are an execution environment for Python in our programming. They're simple, intuitive, easy-to-use, and are an important reason why Python is so popular for prototyping and model building. We'll briefly cover the history of project Jupyter and project IPython. Jupyter came out of IPython in the year 2014. You'll then see how we can install the Anaconda distribution of Python on a Windows machine and a Mac OS machine. We'll then see how we can use virtual environments to work with the Python 2, as well as Python 3 kernel on your notebook server. We'll then explore the Jupyter notebook interface to write Python code. We'll work with code cells, markdown cells, use keyboard shortcuts while writing code. We'll then move onto exploring magic functions within Jupyter. And finally, we'll round out this module by exploring interactive widgets on Jupyter notebooks and also embed a few plots and visualizations in our analysis.

Introducing Jupyter Notebooks

If you're in the process of prototyping your data science project or any other project in Python for that matter, the best execution environment, the one that is the most popularly used is Python on Jupyter notebooks. Jupyter notebooks offer an interactive environment within the comfort and familiarity of your browser. You can write code and view the results of your execution in the same screen as the code itself right within your browser. Python on Jupyter notebooks has dramatically democratized analytics. Jupyter notebooks are an open source web application that allows for interactive development in Python, not just for Python, other programming languages, such as R, are also supported. And Jupyter notebooks come along with the Anaconda distribution of Python. Jupyter notebooks are very popular because they are highly accessible. They are by far the easiest way for a novice programmer to get up and running with programming. They're interactive, they have the read-evaluate-print loop for instant feedback. You write two lines of code, you see the results, then you move on, and they're also extremely powerful. They have a bunch of features to make programming easy. They're ubiquitous. All cloud-based platforms support some form of Jupyter notebooks to support data exploration, data preparation, and analytics. Jupyter notebooks offer you a web-based interactive environment. You write code right within your browser. You have code cells and markdown cells. Code cells are where you write your Python commands. Markdown cells allow you to specify rich explanations and comments in the form of HTML. You can embed your visualizations right within your Jupyter notebook. You can run shell commands and interact with your file system and your operating system from within Jupyter. Millions of people today prototype their Python code on Jupyter notebooks. How did Jupyter come to be? Jupyter notebooks are offered as a part of project Jupyter. This is a non-profit open source project which originated from the IPython project in the year 2014. Project Jupyter includes Jupyter notebooks, as well as the new JupyterLab environment. Before the year 2014, Jupyter notebooks used to be a part of IPython. This is the original project for interactive web-based Python notebooks. When you read documentation on the web, you'll still find references to IPython. Since 2014 though, IPython is a part of Jupyter focused on interactive Python. Recently in the context of Jupyter notebooks, you might have heard of JupyterLab. This is the next generation notebook interface from Jupyter. It's fully compatible with classic Jupyter notebooks, but it has a much nicer user interface with enhanced usability. When you create Python notebooks within Jupyter or JupyterLab, they are essentially ipynb files. Ipynb is the document extension for all of your Python notebooks and it contains all of the content within the notebook. This is not a Python script file, it's a special file that includes all of your notebooks contents, the input code that you've written, the results that are displayed on screen, any images

that you might have embedded within your notebook, any visualizations that are embedded. This also includes the markdown explanatory text in the form of HTML. The ipynb extension comes from the original name for Jupyter notebooks. They used to be called IPython notebooks. This if, of course, from before IPython was integrated into project Jupyter. IPython explains the extension that Jupyter notebooks use today as well. Now these notebooks are stored in the JSON format under the hood. You can't use these notebooks directly as Python script files. You can export these notebooks to the Python script format though as we shall see in the demos.

Demo: Windows Installing Anaconda and Running Jupyter Notebooks

In this demo, we've installed and set up the Anaconda distribution of Python on our Windows machine, and we'll run the Jupyter notebook server in order to execute Python code on notebooks. The Anaconda distribution of Python is available here at anaconda.com /distribution. Along with Python, it has a number of other data science packages as well and it's by far one of the most popular distributions of Python to work with. If you scroll down to the bottom here on this page, you'll find options to install Anaconda for Windows, the Mac OS, and Linux. I'm going to go ahead and select Windows because that's what we are installing right now. The latest version of Python available at this time is Python 3.7. I'm going to go ahead and download the installer, which will now be saved on my local machine. Hit Save and wait for the download to be completed. It might take a minute or so. Once the installer exe file is on your local machine, you can right-click and hit Open and this will bring up a wizard that will walk you through the installation steps. Hit Next, here is the license agreement. You can simply agree with this. You can choose to install Python just for yourself or for all users of this machine. If you have admin privileges, go ahead and hit Next. Here is where the Anaconda 3 distribution will be installed. That's okay with me. Hit Next and go ahead and hit Install. I'm also going to register Anaconda as the system Python 3.7. Once the installation is complete, you see this green line go to the very end, go to Next, hit Next once again, walk through the rest of the wizard, and finally, we are at the last page here. Hit the Finish button and you have Anaconda installed on your local machine. You can head over to the Start menu button, and if you type Anaconda, you'll see the different tools that are available. You have the Anaconda Navigator, which is the graphical user interface, or you can choose to work with the Anaconda Prompt. The Anaconda Prompt is what we'll choose here because this will allow us to run Jupyter notebook from a terminal prompt. This opens up the Anaconda terminal prompt. I'm going to cd into my working directory, which is the Pluralsight folder, and I'm going to run the command Jupyter notebook. This will run the notebook server

and a browser window should open up automatically. If it doesn't, you can simply copy this URL over and paste it into a browser window. This will open up the Home page of your Jupyter notebook within your browser. We have no notebooks available at this point in time. You can click on the New drop-down here and choose the kernel or the execution environment, which for us is Python 3. The Anaconda installation for Windows has automatically installed other kernels or execution environments that you can use as well, such as PySpark, Spark with R, Spark with Scala, and so on. We'll choose Python 3 as our execution environment and this will open up a new notebook on a new tab. This notebook is now ready for you to write and execute code.

Demo: Mac OS Installing Anaconda and Running Jupyter Notebooks

In this demo, we'll see how we can install the Anaconda distribution of Python on our Mac machine, and once we have that, we'll run the Jupyter notebook server that comes with this distribution of Python. In order to install Anaconda for the Mac, you can head over to this URL directly, anaconda.com /distribution/#macos. The Anaconda distribution of Python installs not just Python on your machine, but all of the other packages that you need for data science along with Jupyter notebooks. Scroll down to the bottom here. Here is Python 3.7 and 2.7 available. We'll select the Download option on Python 3.7 because that's what we want to install on our machine. You might have to wait around a minute or so for the package to be downloaded and set up. Once the download is complete, you can right-click and show the package in Finder. You can then double-click on this in order to get started with the installation. Installation is straightforward. This will bring up a wizard that will walk you through the various steps that are involved. Hit Continue on every step. Once you come to the license agreement and the Read Me, make sure you read it and that you're okay with it. Hit Continue. Here is the license. Hit Continue and agree with this license agreement. The next step is to specify the installation type. Anaconda will be installed only for the current user, not for all users of my machine. Wait for the packages to be downloaded and installed. This might take a few minutes and then hit Continue. We don't want the PyCharm IDE at this point in time. Go ahead at the Summary page. You can hit Close and you're done. Anaconda has been set up on your machine, and you can move the installer package to trash. If you head over to the applications folder on your Mac, you'll find an icon there for the Anaconda Navigator. This will bring up the graphical user interface that you can use to launch Jupyter notebooks. You have the option here to run the latest version of Jupyter notebooks. That is JupyterLab or we'll go over the classic notebook server. Go ahead and hit Launch and this will bring up a terminal window with the notebook server running. A browser will open up automatically, or if it doesn't, you can simply copy/paste this URL into your browser. This will take

you directly to the Home page for your Jupyter notebook server. I have no notebooks currently, but I have a number of folders for my current working directory that show up here. If you don't want to use the graphical user interface for Anaconda, you can also launch your notebooks directly from the terminal window. I'm going to hit Ctrl+C here and kill the currently running notebook server. And once you're back on your terminal prompt on your Mac, you can simply run Jupyter notebook. This will launch the server once again and give you the URL that you can open up on your browser. If the browser window opens up automatically, you can simply use that. Copy/paste the URL into your browser and we are back to our Home page screen with all of our folders available. If you click on the New drop-down, you'll see that the Python 3 kernel is available for you to use. Only the Python 3 kernel has been installed. You can't really work with Python 2 using this notebook server at this point in time and that's what we're going to fix in just a bit. I'm going to go ahead and shut down my notebook server, and in the next clip, we'll see how we can run both Python 2 and Python 3 on your notebook server.

Demo: Installing the Python 2 Kernel along with Python 3

In this demo, we'll see how we can set up our Jupyter notebook server to run Python 2, as well as Python 3. Here I am on the terminal window of my Mac machine. This is just my local machine. I have no virtual environment set up at this point and time. Now the steps that we're about to follow involves the creation of virtual environments and the steps are exactly the same on Windows, as well as on the Mac. The one thing you need to make sure when you're working on Windows is that you're working within the Anaconda prompt, which is what we used in our earlier clip. Let's see the current version of Python that I have installed on my Mac. This is the Anaconda distribution of Python which we set up earlier, Python version 3.7 .4. This is the default Python on my Mac machine Python 3. If I just run the Python command, it'll take me into the interactive shell window for Python 3. At this point in time, I don't have Python 2 installed on my machine. I can run python2 --version and you'll see that the command is not found. The Anaconda distribution of Python automatically installs the conda package manager. I'm going to use conda to create a new virtual environment where Python 2.7 will be installed. Run the conda create command. The name of this virtual environment is py27. You can specify your own name if you want to. And the Python interpreter that this virtual environment will use is the 2.7 interpreter. This will install a number of packages within this virtual environment. Go ahead and accept and wait for all of the installations to be complete. You can now use these commands to activate and deactivate your virtual environment. I'll now use the conda activate command to activate our Python 2.7 virtual environment, conda activate py27, and observe that my prompt changes. This is an indication to

me that I'm currently working within the py27 environment. Now if I run conda install notebook, ipykernel, this will install the Jupyter notebook kernel within this virtual environment. Wait for a bit until the installation is complete. There are a number of packages that need to be installed. Hit Yes to proceed with the installation. At this point, we've successfully set up our notebook kernel within this virtual environment. If you want the Python 2 notebook kernel to be available for all users of Jupyter notebooks even when they're operating outside of the virtual environment, you need to manually install the Python 2 kernel using this command that you see here on screen. Once you run this, this kernel specification has been installed and will be available even outside this virtual environment. Let's go ahead and deactivate this virtual environment by calling conda deactivate, and once you're outside, you can run the Jupyter notebook server. This will bring up a server, copy, paste the localhost 8888 URL in your browser window. As usual, this will bring up the home screen for Jupyter notebooks, and if you open up the new drop-down, you'll see that you have both the Python 2, as well as the Python 3 kernel to work with. You can now write code within Jupyter notebooks using Python 2 or Python 3.

Demo: Executing Code in Jupyter

In this demo, we'll create our first Jupyter notebook. We'll start writing some Python code. We'll also explore the notebook interface. Here I am with my Jupyter notebook server running. I have this folder called BuildingYourFirstPythonAnalyticSolution and I'm going to head into that folder. I have a number of notebooks and script files set up already. We'll be using those in just a bit. Head over to the new drop-down off to the top right of your screen. I have Python 2 and the Python 3 kernel available to me. I'm going to select Python 2. You can work with Jupyter notebooks using Python 3 as well, but we'll just play around using Python 2. UnderstandingJupyter is the name of this notebook. All of the notebook related options, such as making a copy of this notebook, saving this notebook, renaming it, will be available within the File menu. You'll write your Python code within these notebook cells. If you want to work with the notebook cell in the edit mode, the Edit menu options will help you. The view menu options will allow you to show and hide the toolbar and the header so you have more room for your code. The Insert menu options will help you create new cells in order to write more code above or below the currently selected cell. The Cell menu drop-down allows you to control cell execution. Do you want to execute cells below the currently selected cell, above the currently selected cell, all of that is here. Your notebook here is currently running the Python 2 kernel, that is the execution environment. If you want to change the kernel that you're executing, switch to Python 3 or you want to restart the kernel so all of the variables disappear, this is the menu option that you'll use. If you want to quickly save your

notebook, you'll hit Ctrl+S or Cmd+S on the Mac or you can just click on the Save icon. Let's get started. We have just one cell. When we open up our notebook, you can click this plus button and new cells will be added. I've added a bunch of new cells which I can now use to write code. Now observe that you can always tell which cell is selected. It'll be highlighted clearly and any operations you perform, such as the cut operation, will be performed on the selected cell. I'm going to go ahead and cut all of the new cells that I've created and I'm left with just one. This is the selected cell. I'm going to hit Esc+A and this will add new cells above the selected cell. Hitting the Esc button gets your notebook into command mode. You can then use the command mode to execute shortcuts. I'll hit Esc and a bunch of As and this will add new cells above the current cell. Now this is my currently selected cell. Esc+DD will delete cells. You can see that the cells are disappearing thanks to our keyboard shortcut. Now that you know how to add and delete cells, I'm going to add a bunch of cells to my notebook and get started with coding. Here is my first line of code. This is in Python 2. This syntax is not valid in Python 3. You write your Python code within a cell and then you can hit the Run button here in order to execute this Python code. The output of your execution will be available to you right under the cell. So I'm going to type in something in this input box, hit Enter, and here is my output. Let's write our next bit of Python 2 code. We'll just use the compare function to compare one and two. Once again, this is in Python 2, and in order to run this code cell, I'm going to click on this run button here. Comparison tells me that 1 is greater. Let's go ahead and write some more code. This is a for loop for x and xrange 1 to 5 print x. Hitting the Run button to execute your code each time is painful. You can simply use the shortcut Shift+Enter and Jupyter will perform the execution for you.

Demo: Restarting and Switching Kernels

Once you've written your code, if you want to insert some other code, you can simply select a cell, go to Insert and Insert Cell Above. This will insert a new cell just above the currently selected cell. Similarly, you can select a particular cell. Here, the comparison cell is what is selected and choose the option Insert Cell Below, and a new cell will be created below the selected cell. You can also cut and paste cells. Select a particular cell. You'll know the one that's selected because of the highlight. Go to the Edit option and choose Cut Cells. The cell that you've selected will be cut. If you want to paste a cell back in, you can select the cell to position your cursor and then go to the Edit menu and see Paste Cells Above, Paste Cells Below, all of the options are available. Observe that the compare code is now pasted above your selected cell. With this compare cell selected, I'm going to head over to the Edit option and got to Cut Cells. This will get rid of the cell and it has disappeared. Now if you want to undo this deletion of this cell, you can go to Edit and

say Undo Delete Cells so the cell that you deleted or cut will come back into your notebook. Once you've written your code, you might want to move your code cells along. Select a particular cell. You can choose the up arrow and move your code cells up. In exactly the same way with a cell selected, you can use the down arrow to move the code in that cell down to the right position. I'm going to go ahead and select the cell that contains my for loop and I'm going to open up this Edit option once again. I'm going to copy the contents of the cell and then with my cursor I'm going to select another cell and then choose the option to Paste Cells Above. Whatever code you've copied will be pasted above the currently selected cell. I'm going to scroll back down and select this for loop that I have here at the bottom, and I'm going to get rid of the cell by choosing the Cut Cell option. At this point in time, the Python kernel is running and executing the code that we have. You can go to the kernel drop-down and choose the Restart option. When you restart the kernel, all of the previous statements that you've executed will be lost from Python memory. Any variables that you've created will be lost. When you hit Restart, you're starting execution afresh. However, you can see that your notebook cells still contain the output of the previous execution. If you want to clear the output of your previous execution, you can head over to kernel and choose Restart and Clear Output. This will restart your kernel. All of the variables that you have created and your current execution will be lost and the output produced will also be cleared. You can see that all of the previous output results have been cleared. If you want to re-execute all of the cells that you have, you can go to Kernel, Restart and Run All. A new kernel will be set up and all of your code will be re-executed from scratch starting from the first cell. This is a quick way for you to re-execute all of the code that you have within your Jupyter notebook, rather than executing the cells one at a time. So far, we've been working with Python 2, but it's quite likely that you want to work with Python 3. You can go to kernel, change kernel, and switch your kernel to Python 3 and observe that on the top right it shows you that your current kernel is running Python 3. I'm going to restart my kernel once again using the kernel drop-down, so I start execution afresh. And once the kernel has been restarted, I'm going to try and execute this Python 2 code within my Python 3 notebook. Select a particular cell, hit Shift+Enter, and you'll see that the execution will fail. That's because raw input is not a valid function in Python 3. Let's try this for loop, select the cell, hit Shift+Enter, and that has failed as well. You can clearly tell that our notebook is now running Python 3. Let's select this last compare command, hit Shift+Enter, and this is also an error. It's not available in Python 3. When we change that kernel, this changed the Python interpreter that our notebook worked with Python. Three commands will be valid. Python 2 commands won't work anymore.

Demo: Exploring Magic Commands

In this demo, we'll continue working with our Python 3 notebook and we'll explore magic functions within Jupyter. I've cleared up all of the existing Python 2 code in this notebook, UnderstandingJupyter, and we are running the Python 3 kernel so we'll be working in Python 3. You can run shell commands from within your Jupyter notebook. You just need to make sure that you precede these commands with an explanation. Python --version shows us that we're running Python version 3.7 .4. We'll use the keyboard shortcut Shift+Enter to execute all of our code. If you want to know your current working directory, you can hit! pwd and this shows you that we're working within BuildingYourFirstPythonAnalyticsSolution. If you want to see a listing of files within this directory, you can run the ls command and this shows you all of the files and folders under your current working directory. If you're working in Python within your Jupyter notebook and you want to install a new library, you can use a pip install command. Pip install matplotlib will install matplotlib and you can use matplotlib from within your Jupyter notebook. It so happens that the Anaconda distribution of Python comes along with matplotlib, but this shows you that a pip install from within Jupyter is possible. Let's import the matplotlib package and take a look at the current version. You can see that we're working with version 3.1 .1. Let's install the NumPy package as well using a pip install. This requirement is also satisfied within the Anaconda distribution of Python. We're now ready to explore all of the magic commands within Jupyter notebooks. Now I want to add in a comment here, but I don't want this to be a Python comment, I want this to be an HTML, and we'll write HTML using the markdown specification. With your cell selected, go to this Code dropdown here and switch it over to Markdown. This will convert your currently selected cell to markdown and this comment will be interpreted as a header too. That's what it means in markdown. Hit Shift+Enter and this will give you HTML embedded within your Jupyter notebook for explanations and any other details that you want to hide. Let's take a simple line here, let's explore magic functions, switch your cell over to the Markdown specification, hit Shift+Enter, and here is our explanatory text in HTML format embedded within our Jupyter notebook. Convert code cells to markdown cells using the keyboard shortcut Esc+M. Magic commands in Jupyter offer powerful features over and about the basic Python syntax. If you want to see all of the magic commands available, you can simply run %lsmagic. Jupyter offers line magic commands, that is magic commands that are applied to a single line. Cell magic commands are proceeded by 2% symbols and these apply to multiple lines within a cell. Typically, you'd run magic commands using the percent as a prefix, but automagic is on, the percent prefix is not needed. We'll continue to use it though so that we know these are magic commands. Distributor notebook is running within a directory called BuildingYourFirstPythonAnalyticSolution. I'm going

to run %cd and this will change my working directory so that I'm now in Users/Loonycorn. This is now going to be my current working directory. All of the operations that I perform will be relative to this directory. Here is a script that I've set up. It's a Python script called myscript.py, which defines a function which calculates the square of the input argument x and returns it. And I run a for loop from 1 to 4 and calculate the squares of numbers. Using a magic function, I can now run this script from within my Jupyter notebook. Make sure you specify the absolute or relative path to the script. We are currently in /users/Loonycorn. Myscript.py is present within the folder, BuildingYourFirstPythonAnalyticsSolution. So I need to point to that script, invoke the %run option, and the script will be executed. The %run magic command can be used to execute scripts. It can also be used to execute Jupyter notebooks. On our home screen of our Jupyter notebook server, I have this ipynb file called sin_curve.ipynb. Now this is the Python notebook that I'm going to execute from within understanding Jupyter. Once again, I can use the %run magic command. Make sure you point to the right location of the ipynb file and hit Shift+Enter and all of the code within your Jupyter notebook will be executed and notice that your matplotlib visualization is also available here embedded within your notebook. This is the same visualization in the sin_curve notebook, that's the code that we executed. Using magic functions,

Demo: Line Magic and Cell Magic Commands

it's also possible for you to pass variables across notebooks. I'm going to set up a new variable called string here. I'm now going to use the %store magic function to store this string value on my notebook server, and I'm going to go ahead and delete the variable within my Python notebook. The string has been stored on my notebook server, but this string variable is no longer available within this Jupyter notebook, UnderstandingJupyter. I'll now switch over to the sin_curve notebook that I have open here. Now this is notebook which knows nothing about the string that we had stored. They can use the %store -r string function to retrieve the string variable. The string variable stored on my notebook server is now available within this notebook. It can print string out to screen and you can see the same value that we had set up in our other notebook. This stored string on the notebook server is available even if you restart your kernel and execute all cells again. So I'm going to restart my kernel and run execution of all cells and observe that the string variable is still present. The string variable that we have stored using a magic function is not part of a particular notebook, it's a part of the notebook server, that's why it's available across notebooks. Let's go back to our UnderstandingJupyter notebook and here is where we have executed our sin_curve.ipynb file. Now if you remember, we've just updated the code within that file. If I hit Shift+Enter, you'll find that the newly-added code is also executed. You have a script

file. We have myscript.py under the folder BuildingYourFirstPythonAnalyticSolution. If you want to, simply load the contents of this file into the current notebook and not just execute that code, you can use the %load magic function. When you hit Shift+Enter, all of the Python code from myscript.py will be loaded into your current notebook. The code is available within this cell. You can hit Shift+Enter on this cell to execute this code. We've explored a few line magic commands that Jupyter notebook support. Let's move onto exploring a few cell magic commands. Cell magic commands are preceded by the double percent sign. When you use cell magic, that basically means interpret all of the lines of code within this cell based on this magic command. The markdown cell magic command says interpret this line here in the form of a markdown comment, not as code. When you hit Shift+Enter, you can see that that line was interpreted as a comment, not as code. Let's take a look at another cell magic function. This is the %%time function. This is a great magic function that allows you to perform simple performance profiling of your code. Whatever code you're running within this cell, time the operation. I've added a sleep statement here within this for loop so it takes a little time to run and you can see that the wall time for this operation was 11 seconds. The wall time is the time as perceived by us, the users. The CPU time, the time for actual execution, as you can see, is much lower. Let's try this once again, but this time we'll use the timeit command. Timeit is also used for performance profiling. This runs your code 100, 000 times by default and then provides the average of the top 3 executions. It also gives you an estimate of the standard deviation of execution. If you want to create a new file and write contents out from within your Jupyter notebook, you can use the writefile cell magic command. Whatever you have within your cell will be written out to your new file. You can check the contents of this new Python script file or another way is to simply execute it from within your Jupyter notebook using the %run magic command and you can see that hi is printed out to screen. And here is one last cell magic command before we move on. We are asking Jupyter to interpret the contents of this cell in the HTML format. Hit Shift+Enter and you can see that our cell contents have been parsed as HTML.

Demo: Exploring Interactive Widgets

In this demo, we'll see how we can use interactive widgets from within Jupyter notebooks. I've opened up a brand-new notebook here called working with Jupyter notebooks and it's running the Python 3 kernel. In order to use interactive widgets embedded with the new Jupyter note, you need to pip install the ipywidgets library. Once this installation is complete successfully, you can set up the import statement for the various libraries that we'll use within this notebook. After we work with interactive widgets, we'll do a little visualization using matplotlib, a NumPy, seaborn,

and pandas, set up the import statements for those. All of these Python packages should already be available to you as a part of the Anaconda distribution of Python. Also, set up the import statements for the utilities that we'll use to set up our interactive widgets. Hit Shift+Enter to execute code. We'll use the pandas read_csv function to read in a csv file into a pandas data frame. The IRIS dataset is what we'll work with. This is a very simply dataset commonly used in machine learning and it's available as a part of the UCI machine learning repository. It's a dataset which contains three species of iris flowers and the sepal lengths and widths and the petals length and widths in centimeters for each flower. Iris Setosa is one of the species. Iris Versicolor and Iris Virginica are the others. Let's go ahead and use the @interact decorator to add interactivity to this custom function here. This custom function will operate on our iris data frame and the input arguments to this function are the column that we want this function to operate on and a threshold x value. The default value for the column is sepal_length and the default value for the threshold is file. We'll use these input arguments and index into our iris data frame and display only those records where the column value has created an x. When you hit Shift+Enter, the app interact decorator will automatically interpret the two input arguments to your custom function and set up the right interactive widgets. It's set up a text box for the string input argument and a slider for the integer input argument. The records displayed here are the records on which our filter function has been applied. Sepal_length greater than five. This corresponds to the default values that we had specified for the column and x input arguments. So you tweak the threshold slider. Our custom function will be invoked and the filtering operation will be performed on the iris data frame. Observe that our threshold is now 6, so only those records with sepal length greater than 6 are now displayed. You're tweaking the input arguments to your function using this interactive widget. Sepal length greater than 7 are the only records displayed here. If you want to perform the filtering operation on a different column, you can change the text within this text box. Instead of sepal_length, specify the name of another column, let's say petal_length and the filtering operation will be performed on the petal_length column. We are currently displaying all iris flowers with petal_length greater than 6. You can use this slider and move it around and perform other filtering operations. There are no flowers with petal length greater than 7, that's why the records are empty here. Now we've basically just scratched the surface of interactive widgets within Jupyter notebooks. There are whole host of other widgets and very complex interactions that you can set up. This is just to give you a taste of what's possible. After having set up this interactive widget, I'm going to go to the file menu here and Save and Checkpoint my current code. An explicit checkpoint will be set up.

Demo: Wrangling and Visualizing Data

In this demo, we'll continue with the same notebook and work with the iris dataset. We'll perform some basic data wrangling using pandas and we'll also visualize our data using matplotlib. We'll continue working with the same Jupyter notebook as in the previous clip. We've already read in the iris dataset. If you want a quick look at how many records you're working with, you can take a look at the shape of the data frame. This tells us that we have 150 records with 5 columns for each record. The describe function on your pandas data frame will give you a quick statistical summary of all of your numeric columns. This displays the count of records, the mean for each column, the standard deviation, and all of the percentile values. In the real world, your dataset might contain null or missing values. You can quickly check for this within your data frame using the isnull sum function. All columns shows 0 indicating that no columns have missing values. Observe how easy and intuitive it is to use the Jupyter interactive environment. Let's check for duplicates in our data. Duplicated .head seems to indicate no duplicates. That's just the first five records, though. Duplicated sum shows us that there are three duplicate records. Let's get rid of those records. I'm going to use iris.drop_duplicates to drop duplicate records from my data frame. This is an in-place operation. My iris data frame will be updated. Run the duplicated sum operation once again and you can see that all of the duplicates have been removed. Jupyter notebooks also allows you to view your plots and graphs as embedded plots or visualizations. Let's use matplotlib to plot a simple histogram. This histogram will display the frequency distribution of sepal lengths across all of our iris flowers. This is a simple matplotlib plot and this will be displayed right within your Jupyter notebook. It's an embedded plot. In this manner, your code and your outputs are together on the same page. It makes everything much easier to follow. Let's view another histogram representation of sepal_widths. Hit Shift+Enter and the histogram will be displayed right below your code cell embedded within your Jupyter notebook. You can set up more complex visuals as well. I'm going to use matplotlib to set up two subplots side by side. The first will contain a scatter plot representation of sepal_length versus sepal_width and the second will contain a scatter plot representation of petal_length versus petal_width. Hit Shift+Enter and you'll get a nice subplot representation with two plots laid out side by side. The interactive widget supported by Jupyter allows you to add interactivity to your matplotlib visualizations as well. Here is our plot function which takes as its input argument the species of iris flower. We'll then perform a filter operation on our original iris data frame to extract only those records which correspond to that particular species. And for the selected species, we'll view a scatter plot representation of sepal_length versus sepal_width. Now instead of using a decorator to add interactivity, I'm going to invoke the interact function. We interact with the plot function that we

just set up and the default value for species is Iris-virginica, that's the initial value. And here is the interactive text box widget correspondent to the one input argument to our plot custom function. The default value is Iris-virginica. And here is a matplotlib visualization of sepal_length versus sepal_width only for Iris-virginica flowers. You can, of course, interact and change your input argument. From Iris-virginica, let's switch over to Iris-setosa and observe that our scatter plot is also updated. Now that we've completed basic analysis and visualization and used our interactive widgets, I'm going to save out the contents of this data frame to a csv file, saved_iris_df.csv in my current working directory. If you now switch over to the tab where you have your home screen open, you'll see that the newly created csv file is available there. Let's go back to the Jupyter notebook that we were working with for one last detail. If you remember, we had created a checkpoint earlier. We had created this checkpoint after we had added in our first interactive widget right before we performed all of the data wrangling with our iris dataset. This checkpoint to save to the date and time. Let's see what happens when we revert to this checkpoint. Before you do this, observe that this warning very clearly indicates that this action cannot be undone, so don't do this if you want to lose your current content. I'm going to hit Revert because I've already backed this notebook up. When you hit Revert, your notebook goes back to the state which existed at the time of your checkpoint.

Module Summary

And with this demo where we worked with embedded plots and visualizations and interactive widgets on Jupyter notebooks, we come to the very end of this module on working with Python using Anaconda. We started this module off by introducing Jupyter notebooks, an interactive web-based environment for fast prototyping and model development. We briefly discussed how project Jupyter came to be. We discussed JupyterLab and the IPython project, the original project from which Jupyter was born. We then saw how we could install the Anaconda distribution of Python for Windows machines, as well as the Mac OS, and we saw that Anaconda comes along with many built-in data science libraries. We also explored how we could use Jupyter notebooks to work with both Python 2, as well as Python 3 kernels. We installed the Python 2 kernel in a virtual environment. We then got some hands-on practice writing code within a Jupyter notebook. We worked with code cells, markdown cells for explanatory text, keyboard shortcuts. We also saw how we could restart kernels and execute all of our cells in one go. We then explored magic functions, which allowed us to interact with the file system from within Jupyter. We then saw how we could use Jupyter notebooks for some simple data wrangling and

visualization. We also explored interactive widgets on Jupyter. In the next module, we'll write Python code using integrated development environments, such as PyCharm, Eclipse, and Spyder.

Working with Python Using Other IDEs

Module Overview

Hi, and welcome to this module on working with Python using other IDEs, or integrated development environments. If you're prototyping code and you're exploring data and you just want to view a few visualizations, well Jupyter notebooks are great, but if you're working with code in the real world on huge projects, integrated development environments are meant for production code. In this module, we'll explore four different IDEs that you can use to write Python code starting with IDLE. IDLE is part of the built-in Python distribution and can be installed on Windows, Linux, or Mac OS machines. We'll then see how we can execute and work with Python code in Eclipse by installing the PyDev plugin. We'll then move onto PyCharm and see how we can use PyCharm to debug and execute our code. And finally, we'll see how we can write Python code using the Spyder IDE. In all of these integrated development environments, we'll work with a very simple Python script that is buggy. We'll see how we can set breakpoints and debug code. Using these IDEs, we'll also see how we can set conditional breakpoints. When you get started programming, you're writing prototypes performing exploratory data analysis. Jupyter is fine for prototyping, but for enterprise scale development, IDEs still matter. When you work with Jupyter notebooks, you are using a development environment, but it's more like an execution environment, not an integrated development environment. For large projects that span notebooks require multiple files and tools, consider an IDE. Python at production scale has become more and more common and there are a number of IDEs out there that you can use to work with Python. In this module, we'll explore four, IDLE, PyCharm, Spyder, and Eclipse.

Exploring Popular IDEs for Python

In this module, we'll work with four different integrated development environments for Python starting with IDLE. IDLE stands for integrated development and learning environment and it's Python's own IDE. When you install the latest versions of Python, you'll see that the IDLE IDE

comes along with it. Installing IDLE is, of course, optional. You can choose not to install IDLE and work with Python on Jupyter notebooks or some other execution environment, but IDLE is available freely for you to use whether you're on a Windows machine, Mac machine, or a Linux machine. The IDLE environment offers a Shell window where you'll execute your code and an Editor window where you'll write code. It's very simple and not very feature rich. Its UI is kind of basic and easy to work with. IDLE is aimed at beginners, not someone who is looking for enterprise grade software. If you've ever written code in Java, you might have used the Eclipse development environment. It's a very popular Java IDE. It's freely available and extensively used. Now Eclipse has the PyDev extension or plugin, which you can then install and use Eclipse for Python development. So if you've already been using Eclipse for development, you're familiar with its interface, you're familiar with all of its features. Well, the PyDev extension for Python is a great way for you to use Eclipse with Python. Eclipse is an enterprise scale IDE, it's heavyweight, it's feature-rich, it has an amazing interface, it has great support for debugging. Eclipse is aimed at professionals who are writing production code at scale. If you've written code in Java, you might have come across IntelliJ, another very popular IDE. PyCharm is another offering from JetBrains, the company which makes IntelliJ. PyCharm is a full-feature, Python-specific integrated development environment and it has basically everything that you'll need to execute and debug Python code. Now PyCharm has paid and free editions. The community edition is free and that's what we'll use in our demos, but if you're working on a large-scale project and you want additional support for SQL and HTML along with Python, well you need to use the paid edition. PyCharm is available for Windows machines, Mac, as well as Linux. PyCharm is heavyweight. It offers tons of features, a very nice user interface, lots of functionality built-in. It's meant for professional users writing production code at scale. And finally, the fourth IDE that we'll explore in this module is Spyder. Now Spyder is a part of the Anaconda distribution of Python, and just like you get Jupyter notebooks with Anaconda, you get Spyder as well, and Spyder works on Windows machines, Macs, as well as Linux machines. Now Spyder is fairly basic and not very feature rich when you compare it with Eclipse or say PyCharm. Also, Spyder specifically targets data scientists, so there are some features which are kind of cool when you're working with data, such as the variable explorer handy for NumPy arrays and data frames.

Demo: Installing and Setting up IDLE

In this demo, we'll explore and work with the IDLE environment in Python. IDLE for integrated development and learning. Here I am on the terminal window of my Mac machine. Let's see what version of Python I have running. If I run Python --version, it gives me Python 2.7. That's the built-

in Python on my Mac. In addition to Python, I also have installed on my machine Python 3 and the Python 3 version that I have is 3.7 .4. I'm now going to head over to python.org /download and get the latest version of Python 3 available. Once I have that, I see that there is Python 3.8 .0. That's what I'm going to download onto my local machine. Once the package has been downloaded, I'll right-click and view this in Finder. I'm going to double-click on this and install Python 3.8 on my machine. We are familiar with installing Python. We can walk through the install steps in a very straightforward and simple manner. Go ahead and read the certificate and verification and license information, hit Continue if you're okay with everything. I'm going to hit Agree here, I do agree with the license information, and then go ahead and hit Continue again. Once I have Python downloaded and installed, you'll see that the IDLE development environment comes in as a part of the default Python installation. That's what I'm about to show you here. Go ahead and hit Install Software and this will start the installation of Python on my machine, the latest version of Python, 3.8. I'm going to close this dialog and then head back to my terminal window. I'll move the existing package to trash, I no longer need it. And with this default installation of Python, whether it's on your Mac or on your Windows machine, comes the IDLE environment. IDLE is available for you to use. Now just for a sec, I'm going to head back to my terminal window and see the version of Python I'm using. Python 3.8 --version gives me 3.8 .0. This is the Python version that I just installed and I'm about to use. Go back to the window where the IDLE environment icon was present and I'm now going to double-click on this icon to open up this development environment. We now have IDLE opened up on our machine. The IDLE environment works with two windows, the Shell window where your commands are executed and the Editor window where you type in Python code. I'm going to use the file option here and open up a Python file, which is on my desktop. This is the price.py file. Once you've selected a Python script, to view within IDLE, the Editor window will open up. You can move the Shell window and Editor window around and place them in a location of your choice. I'll move the editor over to the left and the Shell window is now available to my right. This price.py file that I have open on my editor is on my desktop. The code within this is very straightforward. I'm going to read in a csv file called price.csv, which is again located on desktop. I read this into a pandas data frame called price data and print out the contents of this data frame out to screen. This data frame contains very few records as you have seen just a bit when we execute our code. This dataset does contain missing values. I'm going to use the price_data.fillna function to fill in 0 values for the missing fields. Once I've filled in the missing values with zeroes, I'm going to print out the first five records out to screen once again. I then have this very simple function defined here called get_price_info. It iterates over all of the records in my pandas data frame. Using a for loop, it extracts the total price and the quantity and it then calculates the price per unit, total_price/quantity, and prints

this price per unit out to screen. Once we have this function defined within this Python script, I invoke this function, get_price_info. And finally, at the very end, I'll display a very simple bar graph visualization bar plot of things versus total price. The items that were bought on the X axis, the total price on the Y axis. In order to execute this script, I require pandas and the matplotlib library to be installed. Let's do exactly that. I'm going to switch over to a terminal window. The Python 3.8 version that we are running comes with pip. Let's take a look at which pip we are using. It's the same pip as Python 3.8. I'm going to use pip3 to install the pandas library. Pip3 install pandas will install pandas on my local machine for pip3. I'm going to go ahead and use pip3 install to install matplotlib as well. I've now used the pip package manager to install pandas, as well as matplotlib for my Python 3.8 version.

Demo: Running and Debugging Code with IDLE

We now have everything set up to execute this script and debug it as needed. I have the Editor window off to my left and I have the IDLE shell off to the right. In order to run this bit of code, the price.py file, I head over to the menu, choose Run, and hit Run Module. This will automatically execute my script and the results of this execution will be available in the Shell window off to the right. This is what my data frame looks like. It has three columns, things, quantity, and total_price. The display that you see here is thanks to the two print statements that we have in our code which printed out the first five records in our data frame. And ooh, here at the bottom, you can see that our execution wasn't really successful. There was a ZeroDivisionError: float division by zero. Now if you inspect your data frame, it's pretty obvious why this error occurred, but let's see if we can debug this using the IDLE debugger. Head over to the Debug option and turn on the Debugger. IDLE is now in debug mode and it uses the PDB debugger available for Python. Observe that the Editor window hasn't changed, but the Shell window that we have off to our right has an additional tab. Let's take a look at the shell first and you can see that we are in debug mode. Let's head over to debug control, that is the additional tab that has opened up within our Shell window. The Debug window is what we'll use to view variables and breakpoints and other general debug information. I'm going to right-click on the line where I want to add a breakpoint and select the Set Breakpoint option, and you can see that this line is now highlighted in yellow indicating that a breakpoint has been set here. A breakpoint is a way for you to tell the Python debugger stop execution here so I can inspect my data and see if everything is okay. I'm going to go ahead and add another price point here, but I calculate the price of a unit, total_price/quantity. My initial execution told me that the arrow was somewhere around this line. Let's go ahead and run the module once again, this time in the debug mode. Your Python script will start execution

and immediately stop at the first line allowing you to control the flow of execution. Now from this point onwards, you can run through to the next breakpoint by hitting Go, Step into functions, step over functions, and step out of functions. By default, you'll be able to view the local variables that you have created. You also have information here as to where exactly you are in your execution. We are in the very first line import pandas as pd. And the section below tells us where this file is located and all of the local variables that have been defined, none so far. If you hit the Go button here, your code will execute until a breakpoint is hit. I'm going to go ahead and hit Go and we'll run through to our first breakpoint. Our Python execution is currently at the line which fills in missing values with zeroes, and the only variable that we have defined is the price_data variable. If you want to see the output of your execution so far, you need to head over to the Shell window. Click on the Shell tab and you can see that we've printed out the contents of the first five records of the data frame. You can see that under the quantity column, the row at index three has a missing value, a NaN. We haven't yet executed the code to fill this value with a 0 value. Head back to Debug Control. We are currently at our first breakpoint. You can now hit Go again and the execution will go through until you hit the second breakpoint. We are currently at the second breakpoint and we have a number of local variables. Quantity is 105, total price is 1050. The for loop is at the first row of the Python data frame with an index of 0. If you switch over to the Shell window from the Debug Control, you'll see the results of execution so far. This output printed to screen is the result of our second print function. Observe that the NaN value has been filled in with a 0 in the quantity column. Let's head back to the Debug Control in our IDLE environment and hit the Go button once again. This will continue execution until the next breakpoint is hit. This is the second iteration of the for loop. The index of this row is 1, the current quantity is 78, the total price is 2250. If you want to see the output of the previous iteration of the for loop, we can head over to the Python shell, you can see that in the previous loop when we printed out the price of a unit that was equal to 10. This is from the previous iteration. Our execution is currently stopped at the row with index 1, the second iteration of the for loop. I'm going to hit Go again. Here we are at the row with index 2, no error so far. The previous iteration of the for loop executed successfully. Nothing seems to be 0 here. I'm going to hit Go once again and go onto the next row with index 3. For this row, quantity is equal to 0, and when we do total price by quantity, that's what gives us our ZeroDivisionError. We know this is true. Let's hit Go once again and see this in action. Go will continue execution. We'll actually execute this line. Total_price/quantity and you can see in the Python Shell that this gives us a ZeroDivisionError. The debugger allowed us to figure out exactly where the error occurred. Execution has stopped due to the error the debug controls are no longer available. Now let's head back and clear the breakpoint that we've set. I'm going to clear the first breakpoint. I'm going to clear the second

breakpoint as well using right-click and Clear Breakpoint. Now it's pretty clear why exactly we had this ZeroDivisionError. There are many ways you can fix this code. I'm going to do something very simple. I'm simply going to wrap this code in a try except block. I'm now going to go back to the Run menu and Run this Module. When I click on Run Module, IDLE figures out that the source code has been updated, but I haven't saved my changes. Hit OK to go ahead and save your changes. And because I'm still in debug mode, execution stops at the first line of my code. That's totally fine. I'm just going to hit Go, and because we have no breakpoints set, execution will run through to the very end. We have no errors. Head over to the Python Shell and these are the output results all printed out to screen with no errors. And our final lines of code which displayed a matplotlib graph is displayed within a new window, Barplot of Things Versus Total_Price. In this demo,

Demo: Installing Eclipse and Setting up the PyDev Plugin

we'll see how we can use the Eclipse integrated development environment to run and debug Python code. Getting Eclipse installed on your machine is straightforward. Just head over to eclipse.org /downloads/packages and Installer. There is a download button here. Click this button and Eclipse will automatically detect the kind of OS that you're using. Hit Download 64-bit and you can see that Eclipse has figured out that I'm running on a Mac. Hitting Download here will download a dmg file that I can use to install Eclipse. Right-click, Show in Finder, and you can double-click to start the installation process. The installation process for Windows users should be very similar. Here is my Eclipse Installer, I'm going to double-click on this, hit the Open button here, and get started with installation. I'm going to select the Eclipse IDE for Java Developers. We'll add in the extensions that we need for Python development in just a bit. Specify where you want Eclipse to be installed on your local machine and hit the Install button. Go ahead and accept the license agreement so you can use Eclipse. Once installation is complete, you can select the option to trust Eclipse certificates, accept selected, and keep moving. Once installation is complete, you'll be prompted with a Launch button which you can then click and this will bring up an option to start up Eclipse within a workspace. Go ahead and launch within this workspace. You now have Eclipse installed and running. Eclipse is usually used to develop in Java. It also supports other programming languages and all of this via perspectives. Head over to window and select Perspective and let's open up to see what perspectives are currently available with our default installation of Eclipse. There is a Java default perspective, [00:17: 47.021] the Java type hierarchy perspective, a debug perspective, a Git perspective, but no Python. Let's cancel out of this and head over to Help and select the Install New Software option. This is where we can download and

configure the PyDev extension for Eclipse allowing us to work with Python using this IDE. Specify the absolute path where the PyDev extension will be found pydev.org /update and you can see that Eclipse has pulled in the PyDev extension, which you can then select and hit Next. This will allow you to install the PyDev extension for your Eclipse environment. You can see that PyDev for Eclipse is available here. Hit Next again to continue with the installation process. Hit Finish and accept the terms of the license agreement. You may see a warning which indicates that the extension that you're installing is unsigned. I'm going to go ahead and install this anyway. You'll need to restart Eclipse for the PyDev extension to be enabled. Go ahead and hit Restart now, and once Eclipse has been restarted, you can head over to Window, go to Perspective, go to Open Perspective, and choose Other. Thanks to the install steps that we followed, the PyDev extension is now available to us. This is the perspective I'm going to open up in order to write and debug Python code. And here is our Eclipse workspace in the PyDev perspective. Click on this little icon to the top left and select PyDev project in order to create a new PyDev project and give it any meaningful name. I've just called it My First PyDev project. You can stick with the default settings for project contents and project type. You do need to configure an interpreter for your Python code. Select the configure and interpreter link and choose from the list available. This will scan your local machine and figure out all of the Python installations that you have available. You can select the version of Python you want to use for this project. I'm going to go with Python 3.8. Python 3.8 is going to be my Python interpreter for this project. Hit OK and you're all set up. Hit Finish on this dialog and you're ready to get started coding in Python within Eclipse. Let's create our first Python file within this project. Select the project, right-click, go to New, and File. This will create a new file. You can give it a name. Price.py is the file that I'll continue to use. Hit Finish, hit OK for all of the default options. Here you can configure them if you want to. Here is price.py. We are now ready to get started writing code. Your main Editor window is where you'll write code. I'm simply going to paste in the contents of price.py, contents that we are familiar with, that we've seen earlier when we worked with IDLE.

Demo: Running and Debugging Code with Eclipse

We are now ready to get going with executing our Python code and debugging using Eclipse. From the Eclipse toolbar, head over to the Run option, select Run As, and choose Python Run. This will bring up a dialog, which gives you the option of what file you want to execute. I just have the price.py file and that's what I select. This will execute the Python code within your price.py file, and as you might expect, at the very end you get your ZeroDivisionError. The console here at the bottom is your execution console where your output or results are displayed. You can select and

drag this console to another pane if that's where you want it to be. Now that my console is off to the right, it makes it much easier for me to see what's going on. Let's get started with debugging our code. You can add breakpoints by heading over to your code editor, and right next to line number, you need to double-click and breakpoints will be added. We've added two breakpoints here, one in the fillna line and another within our for loop. You can right-click and perform other operations with breakpoints as well, such as remove breakpoints, disable them, and so on. If you want to run or debug your code, a quick way to access these options is to use these icons here on top. The little bug that you see here will start the debugger and that's what we're going to select. Select the bug icon and Eclipse will start executing this file in debug mode. Debug mode has a different perspective in Eclipse which gives you more information about the current execution context. We'll switch to Debug mode to get this additional bit of information. Eclipse in debug mode has started the execution of this file and stopped at the first breakpoint. In the debug perspective, the console is once again at the bottom here. I'm going to drag this console off to the right so that we can see what's going on. Here is our data frame printed out to screen. Our debugger is currently stopped at the first breakpoint. You can click on this Break tab here in order to view the breakpoints that you have set up. We have exactly 2, 1 on line 8 and 1 on line 19. If you want to see the current state of all of the local variables that you have, you can head over to the Variables tab here and here are the local variables that have been instantiated. I'm going to switch over to the Console tab here so we can view the current state of our execution. You can control the execution of your code using these buttons here on top. Resume will run until the next breakpoint. If you hit Terminate, that'll stop debugging. You can step into functions using the Step Into option here. You can step over lines and execute one line at a time using the Step Over option. You can get out of functions by using the Step Return option. This Debug icon is what you'll use to start debugging. We are already in debug mode. If you click this once again, this will open up another debugger instance and that's exactly what's happened here. That's totally fine. We'll continue debugging using the second instance, which has stopped at the first breakpoint. I'm going to switch over to full screen on my Eclipse, and I'm going to step over this line of code, execute one line at a time. I'm going to keep clicking the Step Over function and execute this Python code until I get into the function get_price_info. I'm going to keep clicking here until I hit my second breakpoint and I run through this for loop exactly once. Here I am at the beginning of the second for loop iteration. The first iteration is complete and the price per unit 10 has been printed out to screen. I'm going to continue clicking Step Over and step over the code that I have one line at a time until the second iteration is complete. The second iteration over our pandas data frame also went through successfully with no errors. I'm going to continue hitting the Step Over button and step over my lines of code one at a time until I get back to the beginning of the

next iteration of the for loop. The previous iteration ran through fine. Here is where I know I'm going to have trouble. I'm going to step over one line at a time, total price, total quantity, quantity is equal to 0. The buttons have been disabled. My program has crashed and you can see the ZeroDivisionError off to the right. Well, this is how you debug using Eclipse. We have some idea of what's going on and what's wrong. I'm now going to set a conditional breakpoint in Eclipse. I'm going to right-click and head over to Breakpoint Properties. Instead of breaking at every iteration of the for loop, I'm going to enable a condition. This condition will basically say break at this line only when quantity is equal to 0. This actually makes my debugging a lot easier because I can specify exactly when I want my breakpoint to be hit. With this conditional breakpoint set, let's head back into debug mode. Click on this little bug icon here to start debugging your code. Our first breakpoint has been hit. Remember, that's not a conditional breakpoint. I'm going to keep hitting the Step Over button here and allow my code to run through until our conditional breakpoint is hit. The conditional breakpoint is when quantity will be equal to 0. The breakpoint that we have within the for loop will be hit only when quantity is equal to 0. That's what conditional breakpoints do. There our program has crashed. Now that we know how breakpoints, conditional breakpoints, and debugging works within Eclipse, we can go ahead and double-click to get rid of all of the breakpoints that we had set in our code. Once all of the breakpoints have been removed, we can go ahead and set up the code to handle the ZeroDivisionError exception. Now that we've handled this exception, you can choose to go back into debug mode or you can simply hit this Run button here in order to run your code. If you run within debug mode, there are no breakpoints and you can see that our code has run through successfully. And the matplotlib graph that we display at the very end opens up on a new window.

Demo: Installing and Setting up PyCharm

In this demo, we'll see how we can execute and debug Python code using the PyCharm IDE. PyCharm is an extremely powerful Python integrated development environment available at jetbrains.com /PyCharm. JetBrains is the same company that makes the very popular IntelliJ IDE for Java. Hit the Download button here on this main PyCharm page and this will take you to the option where you can pick your OS, Windows, Mac OS, or Linux. You can also choose between the PyCharm professional version and the community version. The professional version, of course, is paid. Though there is a free trial available, I'm going to go ahead with the community version for pure Python development. Select the Download option and this will download the PyCharm dmg onto your local machine. I'm going to right-click, view it within my Explorer window, and double-click to get started with installing PyCharm. Here is the PyCharm application which I'm

now going to drag to my applications folder in order to install PyCharm as one of the applications on my local machine. Here within applications, PyCharm is now available. Double-click on the PyCharm icon here. Select the Open option. I do want to get going with PyCharm. Go ahead and hit OK here. I don't want to import any settings. And then either choose to send or not send user statistics. And finally, after all of this clicking, here is PyCharm. I'm going to click Create a new project and set up a new project named my_first_project. Click on the Create button and this will create a new Python project within which we can start working. Close this little pop-up here and make PyCharm full screen. We're ready to get going with our first project in PyCharm. In order to create a new Python file within this project, I'm going to right-click, choose New, and choose Python file. Give your script file a meaningful name, price.py is what we'll continue to work with. Here is our new Python script file within our project. Go ahead and paste the code that we've been working with so far. This is exactly the same code that we've been working with so far. We read in the csv file into a pandas data frame, perform a few operations, and display a matplotlib. Within get_price_info, our processing will throw a ZeroDivisionError that we'll then try and debug using PyCharm. In order to execute our Python script here, you need to head over to the Add Configuration button to the top right. This brings up a dialog and click on this plus icon to add a new configuration, which is to run Python code. Select Python and specify the path to the script that you want executed. This path here is the price.py file within your project. If you have multiple Python versions installed on your machine, you can then select the Python interpreter that you want to use. I'm going to go ahead with the project default Python 3.8. Our configuration is now complete. Hit the Apply button and then hit OK and this will close this dialog. We've configured our editor, but we're not completely done. Head over to File, Other Settings, and Preferences for New Projects. Here is where we'll set the Python interpreter for all new projects that we create. We'll also set up the packages that we'll use within this script. PyCharm only recognizes the pip package and set up tools. Click on the plus icon here at the bottom left and this will bring up a list of packages that you can install. Our script definitely requires pandas to run. So I'm going to select the pandas package and you can go down to the bottom and specify the pandas version that you want to install and use to execute your script. We'll stick with pandas 0.25 .0. Click on Install Package and wait for pandas to be installed. This might take a minute or so. Pandas has been installed successfully. We still need matplotlib though, so I'm going to click on the plus button here again. This will bring up the list of packages and I'll search for matplotlib and click Install Package. I'll just go with the latest version that is available. I've successfully installed pandas, as well as matplotlib and any other dependency that they might have had. Click Apply and then hit the OK button and that should make this dialog disappear.

Demo: Running and Debugging Code with PyCharm

We are now ready to execute our Python code. Click on this little run icon off to the top right of your screen and this will start execution. The results of your execution will be available within this console window here at the bottom of your screen. Once again, let's scroll down below and here is our ZeroDivisionError. We are now ready to debug using PyCharm. In order to add breakpoints to your code, you need to single click right next to the line number where you want the breakpoints to be added. Once you've added your breakpoints, off to the top right of your screen right next to your Run button, you'll find a Debug button. Click on the Debug button and this will stop execution at the first breakpoint. Within the Debug console at the bottom, you find a little window which shows you what variables have currently been instantiated. We have the price_data variable set up holding our data frame. You can use the little icons to control the flow of your execution. You can show the execution point, Step Over or F8 is what you use to step over lines of code. If you want to step into a function, you can hit F7 or press Step Into. If you're within a function and you want to step out of that function, you can use Shift+F8. You can also run to your cursor. Execute code until your cursor point has been reached. Here is our price_data variable. Observe that PyCharm has correctly understood that this is a data frame. Off to the right here, you have an option to view as data frame, and if you click on this, you'll be able to see the data frame in its entirety. You can see that at this point in time, the quantity for the row with index 3 is a NaN or not a number. Let's close out of this. Our current execution is at that line of code which performs the fill any operation. I'm going to step over this line of code and then go back to viewing my data frame. The fill any operation has been executed and you can see that the quantity for the row with index 3 is 0. This is what causes our ZeroDivisionError. Let's close out of this and take a look at a cool feature that PyCharm offers. You can select a particular variable, right-click, and go to Evaluate Expression, which will then allow you to set up expressions involving that variable. You can simply click Evaluate and the current value of that variable will be displayed to you. You can also change the evaluation expression. If you want to check to see whether there are any null values, you can invoke isnull .sum, hit Evaluate, the expression will be evaluated, and the results will be displayed. You can view this as a pandas series object. This is the result of the isnull sum operation. There are no null values in our dataset. We filled them in with zeros. Go ahead and close this dialog. I'm now going to click on this little icon here that'll run my code until the next breakpoint has hit. This is the line at where we calculate the price of a unit. We are in the first iteration of our for loop. This row executes fine. Quantity is not 0. I'm going to click on this icon once again and we'll run through until this breakpoint is hit again. This is the second row at index 1. This is fine as well. You can go ahead and view the intermediate information in the

form of a series object if you're curious to see what the current value of the row is. I'm going to go ahead and hit that little button, which executes my code until the next breakpoint is hit. We're at the row with index 2, this is fine as well. Hit this icon once again. Here we are at the row that is problematic, which causes my ZeroDivisionError. Go ahead and view the current row in the form of a series object and you can see that quantity is 0. Alright, we know we're going to have trouble here. Hit the little icon to run to cursor and this time your program will crash. We have our ZeroDivisionError. The 0-division exception is clearly displayed within our variables window. You can hit the Run to Cursor once again and this time your program will terminate and all your variables have disappeared. We've seen that PyCharm makes it very easy for you to debug your code. I'm going to right-click on my breakpoint here and set up a condition, quantity equal to 0 is when this breakpoint will be hit. This is a conditional breakpoint. Go ahead and hit the debug icon here and get started debugging. We've hit our first breakpoint here. This was not a conditional breakpoint so it definitely will be hit. Hit the run to cursor icon here and this will run execution through until we get to the row with index 3. This is where quantity is equal to 0. The conditional breakpoint was only hit when the condition quantity equal to 0 was met. Run to cursor once again. This results in the exception being thrown, and when you hit run to cursor once again, your program will be terminated. We've completed debugging. I'm going to show you one last cool feature here in PyCharm before we call it a day. Select the function here, get_price_info, right-click, choose Refactor, and Rename. You can rename your function here within this dialog, call this get_info, instead of get_price_info. When you hit Refactor, you'll see here within this refactoring preview all of the invocations of this function that will be renamed. All of the references to this function within your Python project will be renamed when you hit Do Refactor. We have just the one invocation of this function. You can see the function itself has been renamed and the invocation has been renamed as well. Let's go ahead and get rid of our ZeroDivisionError by changing our code to catch the exception that was thrown. And now in order to run this code, click on this little Play icon off to the top right of your screen. The execution of our code was successful and the matplotlib visualization at the very end opens up in a separate window.

Demo: Working with Spyder

In this demo, we'll see how we can work with the Spyder integrated development environment to run and debug code in Python. The Spyder IDE comes as a part of the Anaconda distribution of Python, and we've already set up an in-store Anaconda on our local machine in a previous module. The Anaconda navigator is available here within my Applications window. I'm going to double-click and bring this up. We know that the Anaconda distribution of Python includes

JupyterLab and the classic Jupyter Notebook interface. It also includes Spyder, a complete integrated development environment. Go ahead and launch and open up Spyder. This is the Spyder IDE where you can edit and debug your Python code. Click on this little folder icon here to open up a file present on your local machine. I have the price.py file on my desktop and that's what I'm going to open up and work with. In order to execute this file, all I need to do is hit this big Run button here at the top of my screen. Spyder throws up a dialog asking me to confirm the settings of my execution. I hit Run and my script will be executed. We know that the initial part runs just fine. The errors here at the bottom are ZeroDivisionError. If you want to debug your code, you can hit Ctrl+F5 or go to this little icon here on top which will start the execution of your script in debug mode. The buttons that you see next to the launch of your debugger allow you to control your debug execution that'll execute the current line. This button here will step into a function. This button that you see here will run until the current function or method returns. And then finally, this last icon here will continue execution until the next breakpoint is hit. In order to set a breakpoint, you can place your cursor on the line where you want your breakpoint to be set. Head over to a Debug menu and go to Set or Clear breakpoint or you can simply hit F12. We've set two breakpoints in our code, one in the fillna function and one where we calculate the price of a unit within our for loop. Go ahead and start debugging by clicking on this little debug button. We've hit our first breakpoint, the fillna function. The highlighted line is where your current execution context is, and off to the right of your screen, you can see the current line that's being executed in the IPDB debugger. I'm stepping over lines of code here using this step over icon here on top. Click on this icon repeatedly and step over lines of code. I'm now getting into the get_price_info function and we're going to calculate the rice of a unit. The only button that I'm clicking here is the Step Over button. This will calculate the price of a unit. We are in the very first iteration of our for loop. We'll now print out the price of a unit and 10 is printed to screen. So far so good. The first iteration of the for loop which process the first row in our data frame executed just fine. We are currently in the beginning of the second iteration of the for loop. So I'm going to click on this button here which says continue execution until next breakpoint. So we'll execute all of the lines of code until we hit the price of a unit code. That's where our next breakpoint is. I'm going to keep hitting the button which says continue execution until the next breakpoint and you'll see the result printed out to the right bottom window. Click on the button once again. Here is the result of the next iteration, price_of_a_unit, hit Continue execution until next breakpoint, and this is where we encounter our ZeroDivisionError. If you have multiple breakpoints enabled in your code, you can head over to the View option, go to Panes, and go to Breakpoints. This will display all of the breakpoints that you have enabled. Spyder also supports conditional breakpoints. Go to the Debug menu and go to Set/Edit conditional breakpoint. Make sure that the

cursor within your Edit window is placed at the line where you want this breakpoint to be set. I'll specify the condition here, hit this breakpoint when quantity is equal to 0. If you look at the Breakpoints tab, you'll see that this condition is now associated with our second breakpoint. Let's go ahead and start debugging once again. Click on this Debug icon and your code will execute until the first breakpoint is hit. This is our breakpoint with no condition. It'll always be hit. You can now choose to execute just the current line, so you'll step over one line and execute one line at a time. Executing one line at a time is useful because you can see the output of each line within your IPython console. Executing one line at a time brings us to the get_price_info function definition. Once this function is defined, we go to the function invocation where we invoke get_price_info executing one line at a time. We'll not step into the function, instead, we'll directly hit our conditional breakpoint. The first three records in our data frame are fine and those results have been printed out to screen. We've now hit our conditional breakpoint where quantity is equal to 0. This is the line that's going to cause us our problems. I'm going to step over this line and boom, ZeroDivisionError. Our program has crashed. Our program has crashed, an exception has been thrown. Go ahead and continue to hit Step Over until we completely get out of this debug execution context and we get out of this debug context. Once you're done, we can go ahead and double-click to remove the breakpoints that we've added. I'm first going to go ahead and double-click on the very first breakpoint here that we get rid of that breakpoint. I just have the one breakpoint left, our conditional breakpoint. I'm going to double-click on that breakpoint and get rid of that as well. Alright, it's time to fix our code to handle the ZeroDivisionError. I'm going to wrap this code in a try except block, and now with our code fixed, we can go ahead and hit the Run button. This time our Python script will execute successfully. All of the output is present here within our console and here at the bottom is our bar chart as well.

Module Summary

And with this demo, we come to the very end of this module on writing and executing Python code using an integrated development environment. We started this module off by discussing that Jupyter notebooks were great for prototyping, but if you're writing code at production scale, well you need an integrated development environment. There are many IDEs available out there in the real world for Python developers. We explored four specific IDEs here in this module. We started off with IDLE, which stands for integrated development and learning, which is Python's own IDE. We saw that it was simple and not very feature rich. We then moved onto writing and executing Python code using Eclipse. Eclipse is a very popular IDE among Java developers. We saw that the PyDev extension allowed us to use Eclipse for Python. We then saw how we could

execute and run Python code using the PyCharm IDE created by JetBrains, the makers of IntelliJ for Java. We saw that PyCharm was feature rich, heavyweight, and aimed at professional users. And finally, we saw how we could write Python code using Spyder, which comes as a part of the Anaconda distribution. On all of these IDEs, we worked with a very simple Python script which had an error. We saw how we could execute code, view the results of execution, we set breakpoints and conditional breakpoints to debug our code. In the next module, we'll move to the cloud and we'll see how we can work with cloud-hosted Jupyter notebooks.

Working with Python on the Cloud

Module Overview

Hi, and welcome to this module on working with Python on the cloud. Now if you're working with large amounts of data and your organization is on a cloud platform, well in order to crunch this data, you need to run Python on the cloud, and you may use cloud-based notebooks for your prototyping and development. We've seen how easy Jupyter notebooks are to work with, they're also extremely popular, which is why all major cloud platform providers, Microsoft Azure, AWS, as well as the GCP offer some version of the Jupyter notebook for you to work with data. In this module, we'll explore a few of these options available. We'll work with Azure notebooks on Microsoft Azure, which is a completely free offering. We'll then work with cloud Datalab Notebooks on the Google Cloud Platform. This is GCP's proprietary notebook format built on top of the basic Jupyter notebook. And finally, we'll work with Amazon's manage machine learning service SageMaker and run SageMaker notebooks on AWS. When you're processing huge amounts of data, sometimes your local machine isn't enough, which is why running Python on the cloud has become so common. There are important advantages that you get over local execution. Running on the cloud allows you to use specialized hardware, such as GPUs, or graphics processing unit which you can use to build and train deep learning modules. Running Python in the cloud also allows for easy integration with other cloud-based services. You could run distributed training using the SageMaker ML Platform if you're running on AWS. And finally, running Python on the cloud gives you ease of scaling to large clusters for prediction, as well as training your modules.

Jupyter on the Cloud

Many organizations today have comprehensively moved to the cloud, and of you're working on the cloud, it's quite possible that you're using one of the three major cloud providers, Amazon Web Services, Microsoft Azure, or the Google Cloud Platform. Now each of these cloud platforms have different ways that you can run Python. You can run it on hosted Jupyter notebooks, which runs on a virtual machine instance. It's possible that you're running your hosted Jupyter notebook on a deep learning virtual machine, which is equipped with a graphics processing unit or you might be using your notebook to run distributed training on a platform-specific machine learning framework. Across these three cloud platforms, the categories that I mentioned here are not really exclusive. Cloud platforms introduce platform dependencies to drive sticky adoption. So you'll find Jupyter notebooks in a variety of different environments on the cloud, and these Jupyter notebooks will allow you to write code which offers seamless integration with other platform specific services. Let's talk about notebooks available on the three cloud platforms. Now the notebook that I refer to here is a cloud-hosted Python notebook. Now this could be a platform-agnostic notebook, basically Jupyter running on the cloud, or it can be a platform-specific notebook as in the case of cloud Datalab on the Google cloud platform. Datalab, essentially takes the basic Jupyter notebook hosted on the cloud up one level and offers many platform-specific integrations. If you're building neural network models using deep learning framework, such as PyTorch or TensorFlow, you might want a deep learning virtual machine that comes equipped with a GPU. All three cloud platforms have their own offerings for GPU support using deep learning virtual machines, and when you write Python code on these deep learning VMs, all of them have support for hosted Jupyter notebooks. Each of these cloud platforms offer their own platform-specific framework, such as SageMaker on AWS, the Google AI platform on the GCP. This cloud-specific machine learning service allows you to run distributed training and hosting of machine learning models. And in many instances, the cloud-based Jupyter notebooks form a part of the offerings of this managed machine learning service on the cloud. In this module, when we work with hosted Jupyter notebooks on the cloud, on AWS, or the Amazon Web Service platform, we'll use Amazon SageMaker. In order to run hosted Jupyter notebooks on AWS, we need to set up a SageMaker instance. This may or may not be equipped with a GPU. SageMaker notebook instances run hosted Jupyter notebooks using the classic environment, as well as JupyterLab and these notebooks allow you to execute code that integrates with all other AWS services. On the Microsoft Azure platform, we'll work with Azure notebooks, which is actually a free offering hosted Jupyter notebooks running on an Azure VM. Azure notebooks integrates with an Azure machine learning service. Because this offering is free, you can't really

use a GPU when you work with Azure notebooks the way we will in our demo. The managed machine learning service offering on the GCP is called the Google AI platform. This is an end-to-end platform for data science and machine learning and it offers much more than just, of course, hosted Jupyter notebooks. If you want to prototype and develop your modules, Cloud Datalab notebooks is what you typically use. This is Google's proprietary version of the basic Jupyter notebook. In order to work with Cloud Datalab, you'd need to instantiate a specific Datalab VM that runs a Docker container with Datalab and other data science libraries installed. The GCP also supports cloud-hosted Jupyter notebooks running on a deep learning virtual machine equipped with GPUs.

Demo: Getting Started with Azure Notebooks

In this demo, we'll see how we can work with hosted Jupyter notebooks on the Azure cloud platform. This is Azure notebooks, which is available absolutely free for anyone who is looking to use Jupyter notebooks for data science, but doesn't want to install it on their local machine. Head over to notebooks.azure .com. All you need is an Azure account which you can create for absolutely free. This will take you directly to the notebook interface on Azure. Sign in and specify your Azure, username, and password. Once you sign in, you'll be taken to the main dashboard for Microsoft Azure notebooks. You create your notebooks and other analysis within a project. You can click on the My Projects tab here and you'll see that we have no projects available at this point in time. You can go ahead and create a new project or if you look to the right there, there is also the option for you to upload your GitHub repository directly to Azure notebooks. We'll keep things simple and stick with creating a brand-new project. You need to give your project a meaningful name, Azure-hosted Python notebook is what I've called it, and I've made this project public so anyone with the URL to this project can see my code. Go ahead and click on this Create button and this project will be created and listed on your Projects page. You can click on this project and get into your Project view. There is this little banner here at the top I'm going to get rid of. I'll worry about this survey later. This is your main Azure-hosted Python notebook Project page. Right now, my project is initialized just with a readme. If I want to create a new Python notebook, I can use this drop-down and select the notebook option. You can also create new folders and files from here. This will bring up a dialog that will allow me to name my notebook and select the right kernel. Python 3.6 is what I've chosen. You can also work with R and F# within this notebook. Click on New and new notebook will be created. Once you click on this notebook, the notebook will open up in a new tab and this is the notebook interface that we are so familiar with, one that we worked with on our local machine. These notebooks are free for anyone to use,

they run on a virtual machine on the Azure cloud, and on that virtual machine, there is a Docker container which basically contains all of your notebook's libraries, including the notebook server. When you start up your notebook, you may see this message which says waiting for your container to finish being prepared. You can work with this Azure notebook just like you would with Jupyter on your local machine. If you need additional libraries, you can get them with the pip install. Pip install pandas will show you that pandas is already available. All of the commonly-used libraries for data science, part of the py data stack, will be available. Let's head back to our main project feature and I'm going to use this upload option to upload a file from my local machine, from my computer. Select the Choose files button here, this will bring up a dialog on your local machine, and I'm going to head over to the BuildingYourFirstPythonAnalyticsSolution folder and upload this price.csv file. Hit the Upload button and this price.csv file will now be available on your Azure notebook project and you can work with this file just like you would with the file on your local machine. Here is price.csv. Let's head back to our AzurePythonNotebook. The price.csv file is in my current working directory. I'm going to import the pandas package and use pandas to read in the contents of this file into a price data frame. Let's take a look at price_data.head and here is all of the information available. As you can see here, working with Azure notebooks is no different than working with Jupyter on your local machine, but now that your notebook is available in the cloud, it makes it much easier for you to share your analysis and collaborate with your team.

Demo: Analyzing and Visualizing Data on Azure Notebooks

Here we are on the main page for our project on the Microsoft Azure notebooks platform. On my local machine, I already have a notebook which I've used to perform some data wrangling and visualizations. I'm going to upload this from my local computer onto the cloud platform. Clicking on the Choose files button here will bring up an Explorer window on your local machine. I'm going to select this Jupyter notebook. WorkingWithJupyterNotebooks is the name. This is the notebook that we've used in a previous module. Hit the Upload button, click done, and you'll be able to work with this notebook and all of the analysis that it contains within your Azure notebook project. Now if you remember, this notebook uses a csv file as well, iris.csv, so I'm going to upload that csv file from my local machine. I follow the exact same process. Choose files will bring up a Finder window on my local machine. I'll select IRIS.csv and upload this to the Azure cloud as well. Once you hit the Done button, you'll see that this file is also available within the current project. With our cloud setup looking more and more like our local setup, I'm ready to work with this Jupyter notebook. I'm going to open this up in a new tab and you can see that the code is exactly the same code that we had written on our local machine. You can scroll down, take a look at all of

your visualizations, everything seems to be in there. I'm going to go to the kernel drop-down and restart and run all cells on Azure. I've made the assumption that the basic data science libraries that we've used are all available on the Azure cloud, and yes indeed, they are. Wait for a bit and all of your code cells will be executed without errors. You can see that the last few cells are currently executing. A few seconds more and we'll be all done. All of our code ran successfully on Azure notebooks. Our notebook contains a number of embedded plots or visualizations. Azure notebooks have this really cool feature called a slideshow. Go to View, Cell Toolbar, and Slideshow and you can convert the visualizations within your notebook into a slideshow. There is some tedious work to do here. For each cell, you need to specify the slide type. If you don't want a particular cell's contents to be part of the final slideshow, you can simply say skip. So for all of the code cells, I'm going to assign this skip option. Those are not going to be part of my slideshow. For all of the visualization cells, I'll include it as a slide. So skip over all of these import statements, skip over reading in your pandas data frame, skip over all the pure code cells. All I'm doing right now is scrolling down this notebook and skipping over all cells which I don't plan to include as a part of my slideshow. This part is rather mechanical, so I leave that to you to perform on your own. Skip over all the data wrangling operations until we get to the interesting bit. Here is our first visualization. I'm going to set the slide type to be of type slide. This histogram will definitely be part of my slideshow. I'm going to scroll down and here is another histogram and I set the slide type to be slide once again. I'll scroll down further and I want the side by side comparison to also be included in my slideshow. And finally, here below is our last visualization. Our interactive plot, this also I convert to type slide. And the final line of code where I save out my data frame I'll just skip. Observe that there is this link on top called Enter/Exit RISE Slideshow, click on this link and your slideshow will be up and running. I'm going to zoom out a bit so it's easier to view. All of the embedded plots within our notebook are available in the form of a slideshow and you can use this little arrow here at the bottom right in order to move forward and backward through this slideshow. Here is our next slide. This is our second visualization. I'm going to continue using that arrow and view all of my plots, including the code in the form of a slideshow. I think this is a rather neat feature that Azure notebooks provide. I'm going to cancel out of this slideshow, that'll take me back to my main notebook, and I'm going to head back to my main project page. Here we are on our main project page. We are done exploring Azure notebooks. Observe this little note here which says running on free compute. If you're done, you can go ahead and stop the container and the VM instance and shut down your current project. You don't need to do this on Azure for Azure notebooks because it'll be automatically shut down if you're not using a project for a while.

Demo: Setting up and Connecting to Cloud Datalab on the GCP

In this demo, we'll see how we can work with cloud-hosted Jupyter notebooks on the Google Cloud Platform. We'll work on Google Cloud Datalab which hosts its own proprietary notebook format that interfaces exactly the same as Jupyter notebooks and they are compatible. Head over to console.cloud .google .com. You'll need a GCP account with building enabled in order to work on this demo. Go ahead and sign in with your username and password. Here we are on the main GCP dashboard. Now when you work on the GCP, all your resources are provisioned within a project. You can click on this drop-down here and create a new project for you to work with. Click on the New Project link and specify the name of your project and hit the Create button. A new project will be created for you and we can provision all of our resources within this project. I'm going to switch from the Cloud Datalab project, which is our current project, and select loonycorn-cloud-datalab as our project to work with. The hamburger icon on the top left will bring up a navigation menu that you can use to access all of GCP's projects and services. Head over to APIs and services and go to dashboard. In order to create Cloud Datalab virtual machine instances that will host our notebooks, we need to enable a few APIs. The first API is the Compute engine API. This is basically the API that allows you to provision VM instances. Select the compute engine API and click on the big blue Enable button. This will enable the compute engine API allowing you to provision virtual machines. Once this is complete, let's go back to APIs and Services, and Dashboard. We're going to enable one more API and this API is going to be for cloud source repositories. These are source repositories available on the Google Cloud Platform just like GitHub and cloud source repositories are needed to work with Datalab because your Datalab notebooks are automatically backed up to these source repos. Click on the Enable button here and Cloud Source Repositories API will be enabled. In order to create a virtual machine instance running Cloud Datalab, I'm going to use the command line and the command line that I'll use is this Cloud Shell command line which allows me to work with the terminal window within my browser. Cloud Shell is a terminal window accessible on an ephemeral virtual machine provisioned on the GCP. The GCP does this for you automatically so that you have a command line shell within your browser to work with. Cloud Shell also works within a certain project. You need to invoke a gcloud config set project project id in order to specify your current project. If you want to know what your current project's id is, head over to the dashboard of your main web console and click on this drop-down here on top. This is where you can access all of the project's associated with this account. Observe the id next to the loonycorn-cloud-datalab project, copy this over, and paste it within your Cloud Shell window. I have a full screen version of CloudShell running on a different tab and that's where I'll head over to and paste your project id in here and

use the gcloud command line utility to set a current project. The gcloud SDK on the Google Cloud Platform contains a bunch of utilities that you can use to work with the GCP. The gcloud SDK is automatically available within your Cloud Shell terminal window, which is why it's so easy to work with this command prompt rather than a command prompt on your local machine. With my current project set, I'm going to run an export PS1 command to change the default prompt so I'll have more room to write my commands. This is what my new prompt looks like. If you want to make sure that you have updated versions of all of your gcloud components, run sudo gcloud components update. Everything seems to be up-to-date here. I'm now going to ensure that I have the datalab utility installed within my Cloud Shell window, gcloud components install datalab. Datalab is already available and we can use this command to create a new datalab instance, datalab create Python-datalab. Python datalab is the name of my virtual machine instance. This instance will run a Docker container with datalab, which contains Python and a number of other data science libraries. You need to provision your virtual machine instance within a certain zone. I've chosen the zone asia-south1-b because it's close to where I'm located. My connections will be just faster. So I've selected zone 36, hit Enter, and wait for datalab to be set up. The provisioning of this VM instance and the installation of the datalab libraries might take a minute or so, so you'll need to be a little patient. Once you see this message here, the connection to datalab is now open, your VM instance has been provisioned, and you can connect to datalab. Go to the top right corner and select the little web preview icon and Change port. Your datalab instance can now be connected with on port 8081. Specify the port number and click on change and preview and this will open up a new browser tab with a very familiar interface. In addition toward Jupyter-like home screen, you can see that a number of folders have already been created for you. Here, there are a bunch of samples showing you how you can work with GCP services from within datalab. There are a bunch of sample notebooks for you to play around with. I leave that to you to explore on your own. We'll create our own new notebook.

Demo: Building a Simple Regression Model on Datalab

Datalab has been built on top of Jupyter notebooks, which is why their interfaces are almost identical. Click on the + Notebook command here in order to create a new notebook. This notebook has just one cell. If you want to create new cells, place your cursor on the first cell and hit Shift+Enter a number of times, this will create new cells for you. If you go to the top right, you'll see the kernel that you're running. The default kernel is python2 at the time of this recording. You can switch over to python3. Your code will now be executed using the python3 kernel. This notebook is currently untitled. I'm going to click on the name of the notebook and

give it a meaningful name. I'll call it LinearRegression because I'm going to use this notebook to perform some very simple regression analysis. In order to perform regression analysis, I need a csv file which is present on my local machine. I head over to the main Google Datalab home page and click on this Upload button here. This will bring up the Finder window on my local machine. I select the weight_height.csv file and I'll click on the Upload button here to upload the contents of this file to my virtual machine here on the cloud. We'll now use the csv file to perform some very simple regression analysis. We perform linear regression using the linear regression estimator object in scikit-learn. The first order of business is to set up the import statements for the packages that you'll use, NumPy, pandas, and matplotlib. The datalab virtual machine comes with this preinstalled. You don't need to explicitly install these packages. I'm going to read in the contents of this weight_height.csv file into a pandas data frame. Let's take a look at the data that we've just loaded. It has gender information, the height of an individual, and the weight of an individual. Let's take a look at how many records exist in this data. Hit Shift+Enter to execute your code exactly like you would on your local machine. There are a total of 10, 000 records. Our regression analysis is going to be very simple. The X variable or the predictor that we'll use is the height of an individual and we'll use that to predict the weight of an individual. Here is our X data, the height of individuals in inches. The target of our very simple regression model using ml techniques is going to be this y variable, the weight of an individual. Weight is expressed in pounds. The scikit-learn framework for machine learning is already installed within your datalab virtual machine. From sklearn.model_selection import, the train_test_split function, which we'll use to split our data into training data that we'll use to train the model and test data that we'll use to evaluate our regression model. We'll use 8, 000 records to train our model and 2, 000 records to evaluate our model once it's been trained. Let's take a look at the y data. The y values are the weights of the different individuals that we'll predict once our model has been built and trained, and we'll build this model using the linear regression estimator object available in scikit-learn. We'll instantiate this linear regression estimator object and assign it to the regressor variable and training this linear regression model to predict continuous values is very straightforward. Simply call fit on your training data, pass an x_train, as well as y_train, and that's it. That's all you need to do to train a regression model in scikit-learn. Our data is simple, that's why we didn't have to do any preprocessing. Let's now use this train model for prediction. Invoke the predict function on your x_test data, and once we have the predicted values from our model, we'll evaluate this model by calculating the r2_score. The r2_score is a standard way to evaluate regression models. R-squared can be expressed as a percentage and it captures how much of the variants and the underlying data has been captured by our linear model. The R square of this model is .85, which is pretty good. Let's see a simple matplotlib visualization of how our linear model fit on our training

data. I'm going to reshape these x and y values so that I can use them with matplotlib, and from matplotlib, I'm importing rcParams to configure the font for this particular datalab instance. Datalab uses slightly different forms. Plot a scatter plot of the training data and the fitted regression line and you can see the original data and our linear model overlaid on top of it. You can also explore some of the other options that datalab has to offer. Under notebook, you can convert your notebook to an HTML format so that it's easier for people to view the analysis that you've done. HTML can be shared far more easily. Go ahead and close this tab where you have the HTML format of your Python notebook open, and let's go back to our main Python notebook and convert this to a Python script. Convert to Python will convert our original document to a Python script that can be executed within a terminal window. Now that we know how to work with datalab, we can close out all of these tabs. I'm going to head back to the main GCP web console, and I'm going to Compute Engine and VM instances, and I'll show you the virtual machine instance Python datalab that is running our datalab notebook server. I'm going to go ahead and close this Cloud Shell terminal window so that more of the screen is visible. If you're done with working with datalab, make sure you clean up after yourself. You stop the VM instance and delete it if you no longer plan to use it. If you've just stopped your VM instance, you can always restart it and all of your datalab contents will be available to you, but if you're planning to delete this VM instance so that you don't incur unnecessary charges, make sure that you download and save any notebooks that you've created to your local machine.

Demo: Setting up a SageMaker Notebook Instance on AWS

In this demo, we'll see how we can work with cloud-hosted Jupyter notebooks on Amazon Web Services. We'll create our notebook instance within the Amazon SageMaker service, the manage machine learning service on AWS. Head over to console.aws .amazon .com. In order to perform this demo, you'll need an AWS account with building enabled because you'll need to provision a virtual machine instance. If you have AWS and you have some free credits, you may be able to use that as well. Sign in with your AWS account and you'll be taken to the main Amazon Web Services dashboard. This main dashboard gives you quick links to all of Amazon's products and services, and in machine learning, you have Amazon SageMaker. Click on this link and this will take you to the SageMaker dashboard. The SageMaker service on Amazon offers a complete platform for you to build and train your machine learning models whether it's using scikit-learn or a deep learning framework such as TensorFlow or PyTorch. The SageMaker service also includes Jupyter notebook instances that you can use to prototype your models. And for data preparation, head over to notebook instances and you can see on this dashboard here that we have no notebook

instance provisioned and running. I'm going to click on this orange button here to create a new notebook instance. Give the instance a meaningful name, a Notebook-on-AWS is what I've called it. The next step is to specify the type of instance you want running your notebook. These are machine learning instance types and you can configure them to have GPU support as well. The ml.t2 .medium has 2 vCPUs, but no GPU and that's what we'll stick with. Whichever cloud platform you're working on, the code that you write within your notebook has the ability to access other services on that cloud platform. This means that you can configure the permissions available to your program when it executes and you do this by assigning a role to your SageMaker instance. I'm going to go ahead and create a new role. You can use one of the preexisting roles as well. Specifically here in this dialog, you can configure what S3 buckets your code has access to. S3 buckets are the object storage mechanism available on AWS. I've selected none here. Any S3 bucket with SageMaker in the name is accessible by my SageMaker code running within a Jupyter notebook. Click on the Create role button and a new role will be created with this set of permissions and this role will be assigned to your SageMaker notebook instance. There are a bunch of other configurations settings that you can specify. We'll just skip over all of that and go to Create notebook instance. When your notebook instance is created, AWS will automatically create an S3 bucket with SageMaker in the name and your code within your notebook will have access to that bucket. Once you click this button, you might have to wait for a minute or so until SageMaker spins up your virtual machine instance and sets up hosted Jupyter notebooks. Once you get the status message that says InService, your notebook instance is now available for you to use. Observe that we have Jupyter, as well as the JupyterLab interface available. I'm going to click on Open Jupyter and this will bring up the classic Jupyter notebook server. If you want examples of code which use the SageMaker machine learning service and access other resources on AWS, you can go to SageMaker examples and a bunch of notebooks are available for you to view. There is a DeepAR notebook here which does something with augmented reality. You can take a look at the preview of the notebook and you can execute the code within this notebook right here on the SageMaker instance. These notebooks are basically SageMaker tutorials showing you how you can work with the service. Head over to the Files page and this is our main home screen. This is where we'll create a new notebook and write our code. Here we are in the

Demo: Executing Code to Integrate with S3 Buckets

Home page of our SageMaker notebook hosted Jupyter notebooks on AWS. Click on the New drop-down and you can see that a variety of kernels are available for you. You can use SageMaker to run Spark, you can use deep learning frameworks, such as PyTorch or TensorFlow. We'll stick

with simple Python 3. Selecting this kernel will open up a new notebook on a new browser tab. Once the kernel is ready, you can write code within this notebook exactly like you would on your local machine. The notebook is untitled as it always is to start off with. You can rename it to something meaningful, Understanding the MNISTDataset. In this notebook, we'll see how we can write Python code to integrate with other SageMaker services, more specifically, S3 buckets. When you're working within the SageMaker machine learning service, you have SDKs that you can use to create a SageMaker session and use that session to connect with other AWS services. The SageMaker session that I instantiate here is my connection with the SageMaker machine learning service from within my notebook. Get_execution_role will get the role associated with this notebook instance. This is the role that we had set up earlier. The permissions to access are the AWS resources that are associated with this role are the permission that our code uses to run in SageMaker. Let's use the SageMaker session to access the current region that this instance is provisioned in. The region is us-east-2. The MNIST dataset is a dataset of handwritten digit images, which is very commonly used in image classification problems, specifically my machine learning students. Every AWS region has an S3 bucket which holds this MNIST dataset and that's what I'm accessing here. This is the training_data_uri for my region, us-east-2. I'm now going to download this MNIST dataset, which is in the form of npy files onto my local SageMaker machine. This highlighted command here will download the handwritten digit images onto my local SageMaker machine, and the second command here will download the labels associated with each image. These are files stored in an S3 bucket somewhere. We simply access them and copy them over to this virtual machine instance, and in our current working directory, we have train_data.npy and train_labels.npy. These are multidimensional arrays, image arrays, and corresponding labels. I'm going to use np.load to load this data into the train_data and train_labels variables. If you look at the shape of the train_data array, you'll see that there are 55, 000 images and each image has a total of 784 pixels. These are 28 by 28 images. I'm now going to use embedded matplotlib plots to plot some of these handwritten digit images inline. I'm only going to display the first 10 out of 55, 000 images, not all 55, 000. So I run a for loop from 0 through 10 and this code here accesses the array corresponding to a single image, reshapes the image so that it's in 2D, 28 pixels by 28 pixels, and displays the image, as well as the corresponding label. And once you hit Shift+Enter on this code, you'll see displayed to screen the MNIST handwritten image dataset, the first 10 images. You can see there is an image of a 7, then there is a 3, and you can scroll down and view the first 10 images. As you edit your SageMaker notebook, if you want to view a diff of your changes, you can use this nbdiff functionality available here. Here are all of the new additions that you've made to your notebook recently. Now let's go back to our notebook. I'm going to close this browser tab, and I'm going to make a single

change here. I've selected this particular line. I'm going to go ahead and cut it so that this bit of code is no longer part of this notebook. And once that is done, I'm going to go to the kernel option and restart and run all cells. All of the cells will be re-executed. Once all cells have been executed, let's head over to this nbdiff here and see the difference between the last version of the notebook and the current version. And the diff here clearly indicates that this code cell which used to exist no longer exists in our current version. Go ahead and close this browser tab. We are done working with SageMaker. You can go back to the main Amazon SageMaker dashboard and make sure you clean up after yourself. Select this notebook and stop this VM instance from running. You can go ahead and delete this VM instance as well. When you're working on cloud platforms, make sure you always clean up after yourselves, otherwise, you'll incur very heavy charges and you won't even realize it.

Summary and Further Study

And this demo brings us to the end of this module on working with Python on the cloud. Now before you get going, make sure you delete the VM instances on SageMaker and the GCP. You don't want to be incurring unnecessary charges. When you work on the cloud cleaning up your resources, those which you don't need should be part of your process. We started this module off with a discussion of how Jupyter notebooks are so easy to work with that all major cloud providers have some offering of Jupyter notebooks on their platforms. These are cloud-hosted Jupyter notebooks. These can be the classic notebook interface, or JupyterLab, or can be some proprietary version as in the case of the GCP. In this module, we explored the three major cloud platforms and their notebook offerings. We saw Azure Notebooks on Microsoft Azure, which was completely free for anyone to use. From Azure, we moved onto the Google Cloud Platform and we saw how we could work with Cloud Datalab Notebooks, a proprietary Jupyter notebook interface available on Google. From the GCP, we moved onto AWS and we worked with the SageMaker notebooks, part of the SageMaker machine learning service on AWS. And on this note, we come to the very end of this course on building your first Python analytics solution. Your learning doesn't have to stop here. If you're a novice programmer in Python and you want some more practice, here is a course on Pluralsight that you might find interesting, Python for Data Analysts. If you want to learn more about Jupyter notebooks and all of the features that they offer, Create and Share Analytics with Jupyter Notebooks is the course for you. Well, it's time for me to say goodbye. That's it from me here today. I hope you enjoyed this course. Thank you for listening.

Course author

Janani Ravi

Janani has a Masters degree from Stanford and worked for 7+ years at Google. She was one of the original engineers on Google Docs and holds 4 patents for its real-time collaborative editing...

Course info

Level	Beginner
Rating	★★★★½★ (27)
My rating	★★★★★
Duration	2h 46m
Released	29 Oct 2019

Share course