DAG C3M1 scripts

Last updated: Sep 28, 2024, 8:57pm

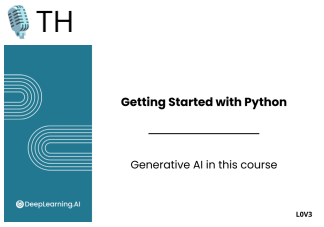
Video title	lecture	Isabel	Sean	slides	alpha
~~L0V1 – Course 3 introduction~~	-	✅	✅	❌	❌
~~L0V3 – Generative AI in this course~~	-	❌	❌	❌	❌
L0V3 – Module 1 introduction	-	✅	✅	✅	✅
L1V1 – Computer programming	✅	✅	✅	✅	✅
L1V2 – Navigating the Jupyter notebook environment (sc)	✅	✅	✅	✅	✅
L1V3 – Input, processing, output	✅	✅	✅	✅	✅
L1V4 – Python or a spreadsheet?	✅	✅	✅	✅	✅
L2V1 – Types, expressions, and printing	✅	✅	✅	✅	✅
L2V2 – Comments	✅	✅	✅	✅	✅
L2V3 – Storing information: variables	✅	✅	✅	✅	✅
L2V4 – Creating lists (sc)	✅	✅	✅	✅	✅
L2V5 – List operations	✅	✅	✅	✅	✅
L2V6 – Taking action: calling functions	✅	✅	✅	✅	✅
L2V7 – State	✅	✅	✅	✅	✅
L2V8 – The coder's mindset	✅	✅	✅	✅	✅
L3V1 – Control flow	✅	✅	✅	✅	✅
L3V2 – Comparison	✅	✅	✅	✅	✅
L3V3 – Branching code: if	✅	✅	✅	✅	✅
L3V4 – Repeating actions: for loops	✅	✅	✅	✅	✅
L3V5 – Branching code: else	✅	✅	✅	✅	✅
L3V6 – Branching code: elif	✅	✅	✅	✅	✅
L3V7 – Execution order	✅	✅	✅	✅	✅

Introduction

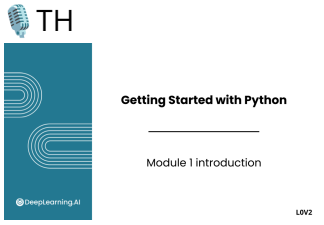
L0V1 – Course 3 introduction

Visual	Script
TH 	Python is just a tool, and it's one way to express your skills as a data analyst. Like statistics, you can spend years studying it and still come away with more each time. Let's take a moment to see how Python fits into your toolbox and set some expectations for what's ahead. In the first course of Data Analytics Foundations, you learned about data & the many forms it can take. You used spreadsheets to analyze and visualize real-world datasets, and implemented the data analytics lifecycle. In the second course, Applied Statistics for Data Analytics, you learned the core statistical methods that power insight-finding. You implemented your skills using complex datasets in a spreadsheet. So what can Python do for you? Data analytics interviews often involve coding tasks in Python. When I'm interviewing, I typically ask applicants questions about statistics and communication, both of which you mastered in the previous two courses. But I also have applicants complete coding exercises in Python as well, to demonstrate that they have the skills to perform a variety of data analytics skills. You'll learn how to do the same in this course. In Data Analytics Foundations, you saw this Venn diagram showing how data analytics sits at the intersection of math, computer science, and business intuition. You learned that data analysts use the field of computing to achieve a specific business goal. In this course, you'll learn that practical application of Python for data analytics, rather than diving deep into computer science theory. By the end, you'll be equipped with the Python skills you'll need to perform data analysis at scale, such as performing statistical analysis and creating visualizations. You won't encounter a palindrome checker here 🙂 That said, think of this course as the start of your journey. Writing algorithms and using advanced data structures are extremely useful. So, expect blah and let's get into it.

L0V2 – Generative AI in this course

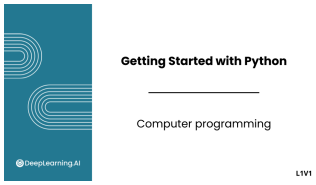
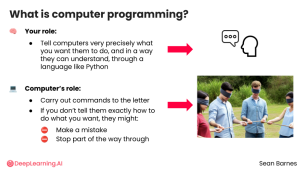
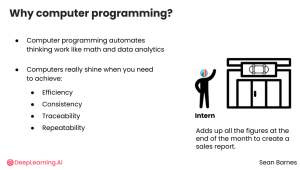
Visual	Script
	

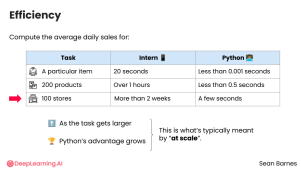
L0V3 – Module 1 introduction

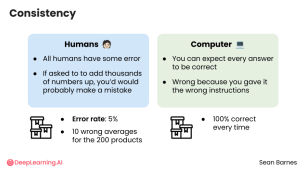
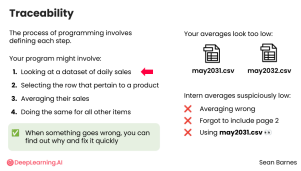
Visual	Script
	Welcome to Module 1: Programming Fundamentals for Data Analytics! In this module, you'll explore the exciting world of Python programming and discover how it can revolutionize your data analysis workflow. Throughout this module, you'll be working with real-world examples, including calculating public library revenue and analyzing restaurant safety inspection scores. This hands-on approach will help you see how these programming concepts apply directly to data analytics tasks. You'll also take a look at how the programs you write compare and contrast with a spreadsheet-based approach. In Lesson 1, you'll learn the role of programming in data analytics, including why Python is such a powerful tool and how it compares to spreadsheets. You'll also get started right away with the Jupyter Notebook environment, where you'll be writing and running your code. Lesson 2 is all about the fundamentals of Python. You'll learn about different data types, how to store information in variables, and how to work with lists. You'll also cover essential programming tools like running functions to perform calculations on your data. By the end of Lesson 1, you'll be able to write your first Python programs to manipulate data. In Lesson 3, you'll learn the core concept of control flow - the logic that determines how your program runs. You'll learn how to make branching decisions in your code using conditionals, and how to repeat actions using loops. These are the building blocks that will allow you to write incredibly powerful data analysis code. I know you'll find coding both challenging and rewarding. Follow me to the first lesson to learn the role of programming in the data analytics workflow. See you there!

The role of programming

L1V1 – Introduction to Computer Programming

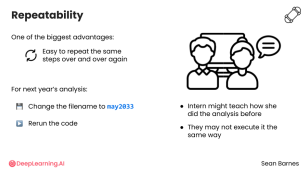
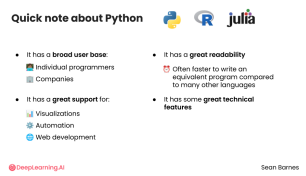
Visual	Script
TH  	Welcome to your first lesson on computer programming! Computers are uniquely powerful at performing a series of instructions that you provide. And there are so many tasks that you can complete
 https://i.ytimg.com/vi/Tgm328zgQqI/maxresdefault.jpg	Computer programming means getting a computer to do your bidding through commands. **[CLICK]** Your role is to **[CLICK]** tell computers very precisely what you want them to do, and in a way they can understand, through a language like Python. Then, **[CLICK]** the computer's role is to **[CLICK]** carry out those commands to the letter.**[CLICK]** If you don't tell them exactly how to do what you want, they might **[CLICK]** make a mistake or **[CLICK]** stop part of the way through. There's a teambuilding game I love where **[CLICK]** four people put on a blindfold while a **[CLICK]** fifth person who's not blindfolded gives them **[CLICK]** instructions for **[CLICK]** creating a square using a rope. You can make the task harder too with different shapes. Can you guess which role is yours versus the computer's? **[pause for thought]** **[CLICK]** You're the one in charge, and the computer is **[CLICK]** blindfolded following your instructions. You can see it's completely reliant on your commands; if the square is lopsided, you are responsible for the failed attempt.
 	**[CLICK]** Computer programming automates thinking work like math, data analysis, and so on. **[CLICK]** Computers really shine when you need to achieve several goals for your analytics work: **[CLICK]** efficiency, **[CLICK]** consistency, **[CLICK]** traceability, and **[CLICK]** repeatability. Say you're working with a **[CLICK]** skateboard shop to analyze their sales. Usually **[CLICK]** one of the interns **[CLICK]** adds up all the figures at the end of the month to create a sales report. Let's see how programming could benefit your analysis in these four ways by comparing it with your intern.

<table>
<tr>
<td> 
</td>
<td>First, let's discuss the concept of efficiency. Say you need to [CLICK] compute the average daily sales for [CLICK] a particular product, like [CLICK] a hoodie. It takes your intern [CLICK] 20 seconds to calculate this statistic using their phone. That doesn't seem like a big deal, but for the same task, Python would likely take [CLICK] less than a thousandth of a second.

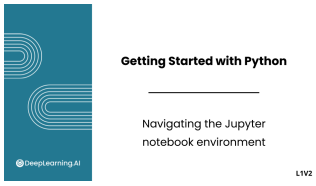
If you have [CLICK] 200 products in stock, your intern will spend [CLICK] over an hour calculating average daily sales while Python gets it done in [CLICK] less than half a second. If you're calculating sales for [CLICK] 100 stores, your intern will spend [CLICK] more than two full workweeks adding up numbers while Python might spend [CLICK] a few seconds. You can see that [CLICK] as the task gets larger, [CLICK] Python's advantage grows. [CLICK] This is what's typically meant by "at scale". Speed is valuable in general, even for a single store, but when you have thousands of stores or millions of customers, that advantage is make or break.</td>
</tr>
<tr>
<td> 
</td>
<td>Next, let's talk about consistency. [CLICK] All humans have some error rate. [CLICK] If I asked you to add thousands of numbers up, you'd probably make a mistake somewhere. [CLICK] Computers on the other hand don't make that type of mistake. [CLICK] You can expect every answer to be correct. By the same token, if a computer's answer is [CLICK] wrong, ultimately it's because you gave it the wrong instructions. If your intern has an [CLICK] error rate of 5%, that translates to [CLICK] 10 wrong averages for the 200 products in the store, while the computer will get them [CLICK] 100% correct every time.</td>
</tr>
<tr>
<td> 
</td>
<td>Now let's consider traceability. [CLICK] The process of programming involves defining each step. In this case, [CLICK] your program might involve [CLICK] looking at a dataset of daily sales, [CLICK] selecting the rows that pertain to a product, [CLICK] averaging their sales, then [CLICK] doing the same for all the other items.

You have an expectation of how each step should work, so [CLICK] when something goes wrong, you can find out why and fix it quickly. If [CLICK] your averages look too low, for example, you might [CLICK] check step 1 and find that it referenced [CLICK] last year's data accidentally, using may2031.csv instead of [CLICK] may2032.csv. You can quickly change the file name and re-run it.

Meanwhile, [CLICK] if your intern's numbers look suspiciously low, it might take awhile to figure out why. Maybe they were [CLICK] averaging wrong, or [CLICK] forgot to include page 2 in their analysis. Or maybe they were [CLICK] using may2031.csv 👀</td>
</tr>
</table>

	Lastly, let's talk about repeatability. **[CLICK]** One of the biggest advantages of computer programming languages is that they make it **[CLICK]** easy to repeat the same steps over and over again. So for next year's analysis, you can just **[CLICK]** change the filename to may2033 and **[CLICK]** rerun the code, assuming you keep your data formatted the same way. That's it! Meanwhile, your intern **[CLICK]** might have to teach the new guy how she did the analysis before, and **[CLICK]** they may not execute it the same way.
	A quick note about Python specifically. Python isn't the only programming language out there. You'll commonly also see languages like **[CLICK]** R or **[CLICK]** Julia used for data analytics. But programming in Python is an incredibly versatile skill for you. Specifically, • **[CLICK]** It has a broad user base, both in terms of **[CLICK]** individual programmers and **[CLICK]** companies where it's used. It's the industry standard. • **[CLICK]** It also has great readability. It's **[CLICK]** often faster to write an equivalent program in Python compared with many other languages. • **[CLICK]** It has great support for **[CLICK]** visualization, **[CLICK]** automation, **[CLICK]** web development, and many other domains. You can really take your Python skills in many directions. • Plus, **[CLICK]** it has some great technical features that make it useful, which you'll learn about in later modules.
TH	Python is the standout choice for data analytics work. In the next video, you'll see the primary Python environment where you'll work with data. I'll see you there!

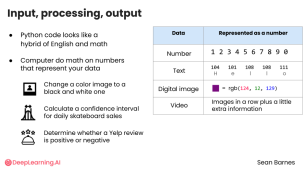
L1V2 – Navigating the Jupyter notebook environment

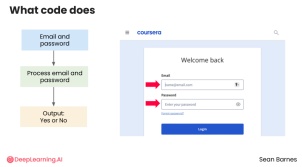
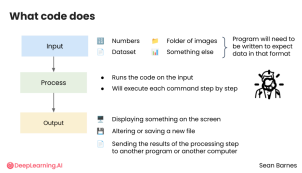
Visual	Script
TH 	In this course, you'll be using Jupyter notebooks for coding, an industry standard tool I've used regularly throughout my career. Let's take a tour of the interface where you'll be coding.
🔗Screencast	This is a Jupyter Notebook. This is where you'll be coding! It almost looks like a Google Doc in a way. It is basically a fancier text editor. You have the name of the file up here in the top left. You have a bunch of options in the ribbon. And then you have this main area where you can edit text and code. The notebook itself is essentially a text document. These different areas of the

	notebook are called cells, they allow you to divide up your notebook. Some of these cells are text cells, and they allow a type of formatted text called markdown. If I double click on the cell, I can see and edit the raw markdown code. So I can add a subheading here, for example. Writing markdown won't be the focus of this course, but it does enable you to create some nice formatted cells in between your code. Pressing Shift Enter will render – or display – the markdown according to its formatting. You'll see markdown throughout the labs in this course. The other type of cell is a code cell, like this one! Running this will print out, "Hi, it's great to see you." Shift Enter runs a code cell and moves to the next one – you'll need to remember this keyboard shortcut; it's the most important command in a Jupyter notebook. That's how you're going to run your code. Here's another markdown cell, and here's another code cell. Don't focus on the code, but if I hit Shift Enter, that will run the code. And the output appears directly below the cell that was just run. You can also click the run button up here in the toolbar. If your code is taking a really long time to run and you need to interrupt it, you can press the stop button. And, if you want to start over with a clean slate, you can hit this restart button and that will reset everything to the way it was before you ran any code. The numbers on the side of each cell indicate the order that you ran the cells. It's not super important right now, but I'm guessing you noticed them. When I run this first cell again, the number changes, and that can just help you keep track of what parts of the code have already been run. This notebook is basically a big text file containing your commands for the computer. It doesn't run the commands by itself. For that you need the kernel. It's a weird word, don't be intimidated by that. The kernel is essentially the engine of the notebook – it is the part of your computer that actually takes the commands and runs them. And when you restart the kernel, you're restarting that engine that's running your code. You won't need to use most of this stuff on the ribbon. One other potentially interesting command here is if you go to edit, you can clear the output of all the cells. Sometimes output can get quite long, and you may want to clear them out. You also have some options in settings as well. You can change to the dark theme if you'd like to, if you want to feel like a cool hacker. And under the theme menu, you can increase the font size of the code or markdown text if you'd like to.

Same screencast as above	This course also includes some reading items that allow you to practice your coding skills. These are optional, but I encourage you to work through them, especially if you are new to coding. It is really through practicing that you will become much more comfortable with coding. In these practice exercises, you'll notice these code blocks that you can run and reset. You'll learn more about variables in an upcoming video, so don't focus too much on the code for now. Complete the exercise directly in the code block and then run the code to check your output. You'll get some helpful feedback on whether your output was correct or incorrect.
TH	Great work navigating the Jupyter notebook interface and code blocks! Jupyter notebooks are quite powerful! Follow me to the next video to see some code examples in action and get a big picture view of what's happening when you run code.

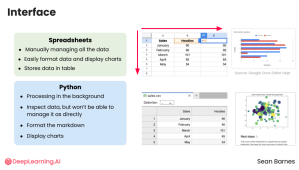
L1V3 – Input, processing, output

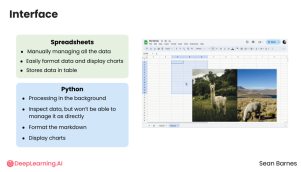
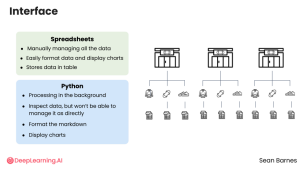
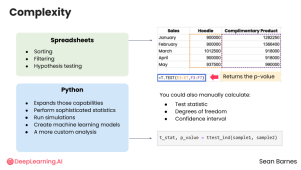
Visual	Script
TH 	It can be pretty mysterious what a programming language does. One useful mental framework for understanding coding is as a sequence of input, processing, and output. Let's take a look.
Slides 	Computer programs basically do math and store results. Now thankfully, your Python code won't be just a bunch of math. It will **[CLICK]** look more like a hybrid of English and math, abstracting away some of the low level computations. That's convenient for you, but under the hood, math is what's happening. It turns out that every kind of data can be represented as a number. • If you're working with **[CLICK]**numbers, well those are **[CLICK]** already numbers. • If you're working with **[CLICK]** text, those can be represented using a **[CLICK]** number for each character. • In a **[CLICK]** digital image, the **[CLICK]** color of each pixel is a number representing how much red, green, and blue should be displayed for that color. • A **[CLICK]** video is just a **[CLICK]** bunch of images in a row, plus a little extra information. • And so on. And fundamentally what computers do is they **[CLICK]** do math on the

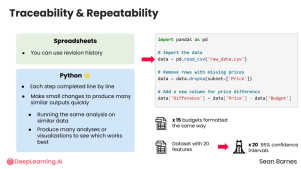
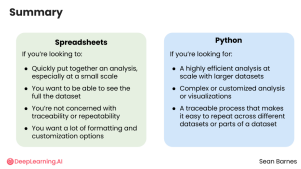
	numbers that represent your data, whatever the format. That math can do basically anything you can imagine, like • **[CLICK]** Change a color image to a black and white one • **[CLICK]** Calculate a confidence interval for daily skateboard sales • Or **[CLICK]** determine whether a Yelp review is positive or negative
	Breaking that down, what code does is it **[CLICK]** takes some kind of information,**[CLICK]** processes it by doing some math, and **[CLICK]** outputs new information. You're used to this type of processing in apps you use everyday. Consider signing into a platform like **[CLICK]** Coursera. Can you think of the input, processing, and output that might happen when you sign in? **[pause for thought]** You're providing it with your **[CLICK]** email and password as an input. Then, a sophisticated computer program **[CLICK]** takes that input and **[CLICK]** processes it. That processing involves several steps like storing your password in a special format for secrecy, sending it over to another computer, and checking whether it matches the information for your account. **[CLICK]** Then, the output of that program might be a simple yes or no! Yes, this is the right password, send this user to their home page. Or no, this isn't right, give them an error.
🔗**Screencast**	Let's look at a couple examples in code. Say I want to add some numbers. Here's a really simple program. It takes three numbers, assigns them to variables, and then adds them together. The input is the numbers 3, 10, and 2, the processing involves adding them together, and, as I hit Shift Enter here, the output involves displaying the result of 15 to the screen. Here's a more sophisticated example you'll explore in the next module. You may remember this movie dataset from the previous course which included the top movies in various years and their durations. I'm going to show you some more complex code here – don't focus on the code, you'll understand each line here by the end of the course. This code will take the entire data set as input, and process it by taking the average duration for each year. The output will be a visualization of this data in a chart. I'm just going to hit Shift Enter to run that, and here's a nice line chart showing the increase in movie durations over the last 80 years or so.
	As you can imagine, each of these components can get quite complex. The input can appear directly in your code, like the three **[CLICK]** numbers you saw earlier, it can be a **[CLICK]** dataset, a **[CLICK]** folder of images, or **[CLICK]** something else. Your **[CLICK]** program will need to be written to expect data in that format. The processing step happens when you **[CLICK]** run code on the input. You've written some commands for the computer, and once you run those

	commands, the computer will **[CLICK]** execute each of them step by step. Processing can be quite mysterious. It happens outside of your field of view, like a magician **[CLICK]** disappearing in a cloud of smoke and **[CLICK]** reappearing a moment later in a different costume. In the movie durations example you saw a moment ago, you never actually saw the movie data. Remember that the computer is just following your commands, and that fundamentally, these are math calculations being performed step by step on your input. Then you have the output. Output can be **[CLICK]** displaying something on the screen, as in the previous examples, it can be **[CLICK]** altering or saving a new file, or it can be **[CLICK]** sending the results of the processing step to another program or another computer. Sometimes you run a program and it seems like nothing is happening! The computer is following your commands, but it might just be that none of the commands involve displaying things on the screen.
TH	There's just one more video before you begin writing code yourself. You've seen the power of programming for data analytics at scale, and how you can run code in a Jupyter notebook to take some input, process it, and give you some output. But when should you write a program versus use a spreadsheet? Both approaches have their commonalities and their strengths. Follow me to the next video to see how to choose!

L1V4 – Python or a spreadsheet?

Visual	Script
TH 	Python and spreadsheets are both digital tools, making them highly efficient and consistent. However, they differ in a few key ways, giving each its niche. Let's take a look at why you might pick one over the other, focusing on the interface, available complexity, and traceability.
	One of the major differences is the interface, as you've seen. With a spreadsheet, you're kind of **[CLICK]** manually managing all the data here. When you want to duplicate a column, **[CLICK]** you have to choose where that column is going to go, and **[CLICK]** manually add the references to other cells. When you're working **[CLICK]** with Python code, this **[CLICK]** processing typically happens in the background. **[CLICK]** You can inspect the data, but you won't be managing it as directly. Relatedly, in a spreadsheet, you can **[CLICK]** easily format your data and display charts. In Jupyter notebook, you can **[CLICK]** format the markdown, and **[CLICK]** display charts, though your capabilities for making the notebook

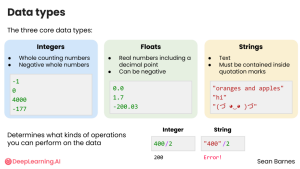
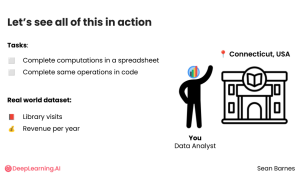
	"pretty" are a bit more limited. Still, these are both formats intended to be shared and explored – to explain data, not just show it in its raw format. Spreadsheets also essentially **[CLICK]** store data in one specific format – a table of **[CLICK]** rows and **[CLICK]** columns. Many types of data aren't best represented in this way.
	You learned previously that unstructured data like text and video doesn't fit neatly into rows and columns.
	Here's another example, say you're dealing with information that has multiple levels or layers. So, if you own **[CLICK]** several skateboard shops, each skateboard shop might have **[CLICK]** a set of products it sells, and each **[CLICK]** product has different sales, that becomes complex to model with a spreadsheet. Ultimately, **[CLICK]** you'll need multiple sheets to model these relationships.
	The complexity of analysis available to you is also different. You have a lot of capabilities in a spreadsheet – from **[CLICK]** sorting and **[CLICK]** filtering to **[CLICK]** performing hypothesis testing. **[CLICK]** Python **[CLICK]** expands on those capabilities. You'll be able to **[CLICK]** perform more sophisticated statistics, **[CLICK]** run simulations, and even **[CLICK]** create machine learning models. Similarly, if you need **[CLICK]** a more custom analysis, you'll have an easier time in Python. Here's an example. In the previous course, **[CLICK]** you conducted hypothesis testing using the `T.TEST` function in Google Sheets. That function **[CLICK]** returns the p value associated with the test. **[CLICK]** You could also manually calculate the **[CLICK]** test statistic, **[CLICK]** degrees of freedom, and even a **[CLICK]** confidence interval. But in Python, you can perform each of these tasks with a **[CLICK]** single line of code, as well as calculate additional statistics for more complex scenarios.
	Python's advantage in available complexity applies to visualizations as well. You may recall that **[CLICK]** it's not possible to create a true boxplot in Google Sheets. That's frustrating, considering how useful box plots are. **[CLICK]** In Python, you can create box plots with just a few lines of code.

	When it comes to traceability and repeatability, Python is the superior choice. If you want to track the steps you took in your analysis in a spreadsheet, **[CLICK]** you can use revision history… and that's kind of it. Your Python code on the other hand will contain **[CLICK]** each step you completed line by line. Here's an example. Say you have a **[CLICK]** raw data file of purchases for a home renovation, and a second version of the file that contains **[CLICK]** cleaner data. **[CLICK]** But what did you do to clean the data? It's like playing spot the difference. If instead you had the Python notebook you used to clean the data, you can examine the steps one by one. **[CLICK]** For example, you might see that you **[CLICK]** imported the data, **[CLICK]** filtered out rows with a missing price value, and then **[CLICK]** added a new column to track the difference between the purchase price and budget. Repeatability in Python goes beyond just reproducing past analyses. You can also **[CLICK]** make small changes to produce many similar outputs quickly. Some common use cases for this strategy are **[CLICK]** running the same analysis on similar data, or **[CLICK]** producing many analyses or visualizations to see which one works best. For example, if you're working with a real estate investor who has **[CLICK]** 15 of these budgets all formatted in the same way, you could **[CLICK]** change 1 line of code – wherever you import the data – then run the same analyses as before with one click. Or, if you have a **[CLICK]** dataset with 20 features, you could construct **[CLICK]** 95% confidence intervals for each feature using minor tweaks to the same command.
Slides 	So to sum up all these differences, **[CLICK]** spreadsheets are the way to go if you're looking to • **[CLICK]** Quickly put together an analysis, especially at a small scale, • **[CLICK]** You want to be able to see the full the dataset, • **[CLICK]** You're not much concerned with traceability or repeatability, • **[CLICK]** And you want a lot of formatting and customization options **[CLICK]** Python will be the stronger choice if you're looking for • **[CLICK]** A highly efficient analysis at scale, that is, with larger and larger datasets, • **[CLICK]** Complex or customized analysis or visualizations, • **[CLICK]** And a traceable process that makes it easy to repeat across different datasets or parts of a dataset
TH	Awesome, so you've seen the fundamentals of how Python programming works, you've seen the Jupyter notebook interface, plus why you'd want to use Python in the first place. Once you've completed the practice assessment for

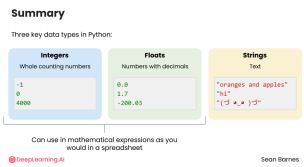
	this lesson, join me in the next one to get hands on with coding. I'll see you there!

The basics

L2V1 – Types, expressions, and printing

Visual	Script
🎙️ TH	Alright! Let's check out the building blocks of code.
Funny face: (づ ◕‿◕)づ 	In Python, you can work with different kinds of data. You can perform different operations on the data depending on its type. **[CLICK]** There are three core data types you'll work with for now: **[CLICK]** integers, **[CLICK]** floats, and **[CLICK]** strings. • An integer represents a whole **[CLICK]** counting number, including **[CLICK]** negatives, **[CLICK]** like **[CLICK]** -1, **[CLICK]** 0, **[CLICK]** 4000, **[CLICK]** -177, and so on • A float, or floating point number, represents a **[CLICK]** real number including a decimal point, and **[CLICK]** can be negative as well. **[CLICK]** Examples are **[CLICK]** 0.0, **[CLICK]** 1.7, **[CLICK]** -200.03, and so on • A string means **[CLICK]** text. In Python, text **[CLICK]**must be contained inside quotation marks. That's how Python knows it's text and not a command. Examples include **[CLICK]**"oranges and apples", **[CLICK]**"hi", or **[CLICK]**this funny face made out of special characters. The type of data is more important than it might look at first glance. That's because, as I said a moment ago, the type **[CLICK]** determines what kinds of operations you can perform on the data. **[CLICK]** Here's an example, can you tell the types of these two values? **[pause for thought - 400 and "400"]** The 400 on the left is an **[CLICK]** integer, while the "400" in quotes on the right is a **[CLICK]** string. Even though they look the same to you and me, the computer thinks of these numbers differently. For example **[CLICK]** dividing the number on the left by 2 will **[CLICK]** output 200, while **[CLICK]** dividing the string on the right by 2 will give you **[CLICK]** an error, since you cannot perform division on a piece of text.
	Let's see all of this in action, in particular what you can do with these data types. **[CLICK]** **[CLICK]** First, you'll complete some computations in a spreadsheet, then **[CLICK]** you'll see how to do the same operations in code. In this lesson, we'll work with **[CLICK]** a real world dataset of **[CLICK]** library visits and **[CLICK]** revenue per year for one library in the state of **[CLICK]** Connecticut, USA.

🔗**Screencast** 🔗**Notebook** 🔗**Public library data set**	Say you get an email from the Plainsville Public Library in Hartford, Connecticut, USA. They've sent you some data here, and they have a few questions for you. Each row represents a year of operation for the library, and each column is some feature of that year. Some of the features include the population of the area served, the number of visits to the library per person, the total income made by the library, and the expenses of running that library. Here's what the library wants to know: the total number of library visits in 2023, income per library visit, total income for both years, and to determine if the library was profitable in 2023. Let's calculate the following: • Total number of library visits in 2023, which is the number of people times the visits per person. In this case, there were over 66,000 visits. • Income per library visit in 2023, which would be the total income divided by the number of library visits. This is in dollars, so about $12.50 per visit. • The total income from both years is the sum of the income from each year. • Was the library profitable in 2023? To figure that out, calculate the difference between the income and the expenses. The answer is yes, the library made about $36,000 in profit that year, which is a currency. So how can you do these same operations in code? What you'll discover is that in Python, these operations are very, very similar to what you're already confident with, having worked in spreadsheets. Here we have the exact same tasks and the exact same data as in the spreadsheet, but this time let's calculate this with Python. First, let's calculate the total number of library visits in 2023. I'm going to start with the print command. Print displays the result of an input to the screen. First I'm going to print some text so I can remember what I'm doing here. Print total number of library visits. Close the quotes, close the parentheses. Now I'm going to add another command that does exactly what we did in the spreadsheet, which is to multiply these two numbers. Now, I don't have a way to reference a cell in the text, so for now we'll just use the numbers. Next, hit enter to start a new line. Then, print the visits per person times the population of the area, 17,479. I'm going to hit shift enter to run this cell. And we get the total number of visits is about 66,400. Great. So just to back up a little bit, print is a command that is going to display the result of an input. In this first print command, I just printed some text so that we can remember the point of the calculation. In the second print command, I entered a mathematical expression for the total number of library visits

	exactly as we did in the spreadsheet. Okay, that's the first task done. Now let's do the second task. In this case, I'm going to print income per library visit, close quotes, close parentheses, and I'm going to enter a new command. We're going to repeat the same task from the spreadsheet: the total income divided by the total number of library visits: 830,696 divided by this number here. So I can just go ahead and copy and paste. Hit shift enter, and here's the income per library visit, 12.51. Great. Let's do the third task, total income from both years. And we'll print some text so we can remember what we're doing here. Hit enter and add the second command, print. Add this number here: that was the income from 2023, plus 763,415 and hit shift enter. There's the total income 1,594,000, which matches what we got in the spreadsheet. Finally, let's determine if the library was profitable. First some text: "Profit in 2023," enter. And this time I'm going to print the total income and just copy that from the previous cell, minus the operating expenses: 794,398. Close the parentheses and hit shift enter. And we can see that the profit was around 36,298. Can you spot the data types that you used in each of these expressions? Well, going back up to the first code cell, the total number of library visits is a string. It's just some text in between quotation marks. 3.8 is a float. It is a number that has a decimal. And 17,479 is an integer. It's a number with no decimal.
	So, to sum up, there are **[CLICK]** three key data types in Python that you'll be working with extensively: • **[CLICK]** Integers, which represent whole counting numbers • **[CLICK]** Floats, which represent numbers with decimals • **[CLICK]** And strings, which represent text And, you **[CLICK]** can use integers and floats in mathematical expressions as you would in a spreadsheet, to do things like calculate profit or income per visit.
TH	Great work! You've seen the core data types you'll be working with every time you code something up in Python. Follow me to the next video to learn the difference between code and comments. I'll see you there!

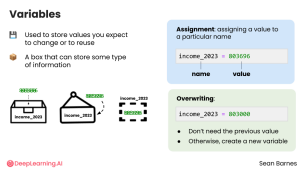
L2V2 – Comments

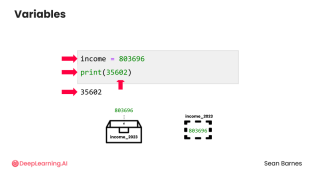
Visual	Script
🎙️ TH	In addition to commands like print, you can also add comments to your code. These act as notes to yourself in the future or to your collaborators.
🖼️ Slides 	Typically, **[CLICK]** comments are intended to clarify your code. **[CLICK]** Comments start with a hashtag, and **[CLICK]** they can be on their own line, or **[CLICK]** they can come at the end of a line that already has some commands. **[CLICK]** When the computer sees the hashtag, it will completely **[CLICK]** ignore everything afterwards and just **[CLICK]** move to the next line.
🔗Screencast 🔗Notebook	Let's say that I'm going to send this code to my colleague, and I want to mark it up with a little bit more information. In this case, I can add a comment. A comment starts with a hashtag. Let's add one here. ~~A comment is text inside your code that is not a command.~~ The computer will completely ignore any information after a hashtag. So it can start at the beginning of the line, it can come at the end of the line. Let's just say this was income from both years. So this is just for me, the person. The computer will completely ignore the commented text. As another example, I can add a comment here at the end that says this should be in dollars, so hashtag dollars. The computer will execute everything up to the hashtag. So if I hit shift enter, and run that again, we get the same output no matter how many comments we put. You can put a comment in between two other lines of code, that's also completely fine.
🎙️ TH	You've seen how comments and code combine. Once you've completed the practice exercises for types, expressions, and printing, follow me to the next video to learn how to store information in Python using variables.

L2V3 – Storing information: variables

Visual	Script
🎙️ TH	As you saw earlier, a key strength of Python is reusability. In particular, reusing data without having to type the same number or string over and over again. Let's take it over to the notebook to see how you can do that.
🔗Screencast 🔗Notebook	You may have noticed there was one particular value that I had to use several times throughout these calculations, and that was total income from 2023. It's kind of annoying to type out this long number so many times, and it creates

	opportunities for mistakes, like typos. Another common situation is, let's say that the library discovered that this number was recorded in error and the actual value was 830,000 even. Right now, I would have to go in manually and retype every single value. And imagine you have this value hundreds of times in your code. How do you avoid this busywork? Let's go back to the original, correct value. What if I wanted to store this value once in order to then use it several times? To do so, I'm going to create a variable called income_2023, and in that variable, I'm going to put the value 830,696. So what I'm doing here is I'm storing this number inside this name. Now, what do you think is going to happen if I print income_2023? Well, let's see... I'll hit shift enter and I get 830,696, the number that I stored inside this name. Cool! Now I can use this value in as many calculations as I would like. I'm going to copy over this cell, where we calculated the profit in 2023, and I'm going to replace 803,696 with income_2023. I'll copy over this cell as well. And I'm going to replace that same number with income_2023. Now what do you think is going to happen if I hit shift enter? Well, I should see five lines print out, one for each print statement. Here we go. So the profit in 2023 matches the same calculation we had before. The total income from both years matches what we had before. And now, if the library realizes that, hey, this number was actually supposed to be 830,000, you can change it in one place, run it again, shift enter, and now it gets updated across all of these results. This process is very similar to what would happen in a spreadsheet. Right now, I have this cell that contains 830,696, and it's kind of like a variable with the name D3. What a variable in this context means is that the value inside the box D3 can change, but, no matter what, this box is always called D3. So in any calculation that involves 830,696, I can use this variable name D3. If I update the value in D3, all of these calculations over here on the right will update to reflect that. For example, if I change this number to 830,000, you'll see that when I hit enter, all of the calculations over here on the right that depended on the value of D3 updated. Let's see what just happened here and how variables work in a little more detail.
Slides Show the three ways side by side, then show them updating	Variables are **[CLICK]** used to store values, in particular values that you expect to change or to reuse. You can think of variables as **[CLICK]** a box that can store some information. This process is called **[CLICK]** *assignment*. Specifically, you're assigning a **[CLICK]**value to a **[CLICK]**particular name.

along with the script. Keep all 3 on the screen. 	Let's visualize variable assignment in three ways, and just see which one makes the most sense to you. I'll use the line of code you saw a moment ago `income_2023 = 803696`, • The first way to think about variable assignment in this line is that you have **[CLICK]** a box, you label it **[CLICK]** income_2023, and you store the value **[CLICK]** 803696 inside it. • The second way is that you have **[CLICK]** a name that means something to you, **[CLICK]** `income_2023`, and you assign the number **[CLICK]** 803696 to that name. • You can also think about this variable as similar to **[CLICK]** a cell in a spreadsheet. You have a cell, **[CLICK]** you store a value of it, and that cell has the name **[CLICK]** income_2023, rather than being a row and column name If this value needs to be updated with **[CLICK]** 803,000 even, then the previous value gets completely **[CLICK]** thrown out and **[CLICK]** replaced. You no longer have the previous value saved anywhere. This process is called **[CLICK]** overwriting, and you should only do it if you are confident that you **[CLICK]** don't need the previous value. **[CLICK]** Otherwise, you should create a new variable.
	There are rules for variable naming. Basically **[CLICK]** stick with **[CLICK]** letters **[CLICK]** uppercase or **[CLICK]** lowercase, **[CLICK]** numbers, and **[CLICK]** underscores, and you'll be fine. Make sure you **[CLICK]** start your variable with a letter. You can read more about the specific rules for naming in the upcoming reading item. **[CLICK]** Keep in mind that variables are case sensitive. The variable `income_2023` is completely different from `Income_2023`. To us as humans it may seem a bit ridiculous, but to the computer lowercase i and capital i are as different as A and Z. Let's take a quick look at creating variables again, now that you're more familiar with them.
🔗Screencast pt 2 🔗Notebook pt 2	Let's create a variable for the total number of library visits. This was the population of the area served times the number of visits per person. What do you think would be a good name for this variable? **[pause for thought]** You could call it something like x... x equals 3.8 times 17,479. This completely works and is fine. You can then hit enter... print x.. and shift enter displays the number of library visits in 2023. But let's go ahead and give this a better name. total_library_visits. It is longer, but it does help a ton to use a good name that actually describes what you're working with. The computer doesn't have an opinion. Then let's print out total_library_visits, shift enter, and it's the same number.

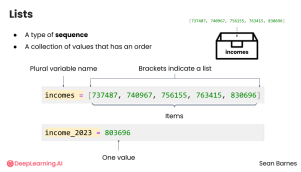
	Now, what do you think would happen if I were to print capital T, total library visits. **[pause for thought]** If I hit shift enter, this doesn't work. You'll see a little bit later in this module how to address these errors, but don't be intimidated! You can see the code right here. And then at the bottom, NameError, Total_library_visits is not defined. So lowercase t... capital T... to you and me, that's the same letter. To the computer, it's a completely different letter. I'm going to remove the second print statement, because it's an erroneous statement. Now, say the library asked you to calculate the total library visits for 2022. What you can do is just make a new line using enter and store a new value inside total library visits. So in this case, I'll say total library visits equals visits per person in the area for 2022, which is 3.3 times... and then the population of the area, which is only different by about 34. I'll copy that, and paste it in here. Now, what do you think will be printed out when I hit Shift Enter? Will it be the same value as before? In this case, if I hit Shift Enter, it's the new value from 2022. That's because these lines happen in order. At first, total library visits is assigned this value on the right, which we happen to know is about 66,000. Then, it's assigned this new value, about 57,000, and that completely overwrites what was in the cell previously. Going back to the spreadsheet from earlier, that's exactly what would happen if I were to change this formula. If instead of having the values for 2023, I decided that I would rather have the values for 2022. Then, whatever was there for 2023 is completely gone, and it's been replaced with the values for 2022.
Slides 	You also saw this calculation `print(income_2023 - 794398)` a moment ago. What's really happening here? **[CLICK]** In this case, you've already assigned income_2023 and now you're **[CLICK]** using it in an expression. When Python comes to **[CLICK]** this line and sees `income_2023`, it looks for a **[CLICK]** box with that name – or you could think of it as a **[CLICK]** cell with that name – and **[CLICK]** replaces the variable name with its current value, in this case 803696. Then, Python proceeds with the subtraction, replacing this entire expression with **[CLICK]** 35,602 and ultimately **[CLICK]** prints out that value as the result of the calculation.
TH	Great work using variables to store values and calculate new ones. You'll use variables in virtually every Python program. Follow me to the next video to get started with a new type of data: lists.

L2V4 – Creating lists

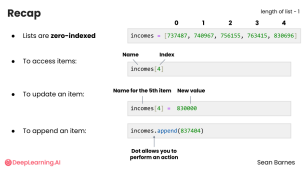
Visual	Script
🎙️ TH 	Single integers, floats, and strings are quite useful on their own, but in data analytics you'll often encounter sequences: ordered collections of values. One of the most common sequences is the list. Let's take a look at why lists are so valuable.
🔗Screencast 🔗Notebook	So, the public library has come back to you with another analysis they want you to do. They want you to sum the total income across all five years. And here's the data right here. They have five different years and the total income made by the library in each of those years. Let me ask you, how would you store this information in Python code? **[pause for thought]** One way you could store this information is with variables, which you just learned about. Following the same pattern as before, I could do income_2019 equals, and then copy this value. And then I would enter income_2020 equals. And then copy this value. And then do income_2021. And then copy this value. This seems inefficient, right? Right now, I only have five years of income. But let me just show you the full extent of this data. I'm going to hop over to my spreadsheet. Here is the data from all of the years of operation for the Plainville Library in Hartford, Connecticut. That's a lot of years: 28 in total. Making 28 variables seems inefficient. One thing you can do instead, and I'm going to create a new cell to show you, is create a list that contains all the incomes. A list is appropriate here because we have a collection of items, they all represent the same thing (incomes), and they are ordered by year. So let's create a new variable called incomes, because it's going to represent many incomes, not just one. This is a variable. Equals, and now I'm going to create a list using the brackets. Brackets denote a list. And now what I can do is take all these values and put them in my list, separated by commas. This is useful because now I have just one variable that contains all the data. I'm going to hit shift enter so that we now have this variable saved and we can use it. I'm going to create new variables for 2022 and 2023 incomes, just to show you the difference between these two approaches.

	So now I have my approach with many individual variables, and I have my list approach. I'll just run both of those code blocks, so I have all those variables available. For the first step of averaging, I want to add up all of the incomes. Well, if I want to do that up here with all my individual variables, I'm going to have to add income_2019 plus income_2020 all the way to 2023, and print that result. Okay, that took a really long time, but the library made 3.8 million dollars. Great for them. How would this work with my list? I can actually use a special function that works with lists. I'll print out the result again. And I'm going to use the sum function, which is also a function like print. I'm going to sum up all of the incomes. What this **[highlighting sum code]** does is exactly this process **[highlighting adding variables]**, but with all of the items inside my list. What output do you think I will get from running this code block? **[pause for thought]** It's the same exact outcome, but which one looks faster and more scalable to you? One last thing before we formalize lists. Let's say the library has come to you and they've found the data from 2017 and 2018. What now? Well in my variables approach I've gotta hit enter, add a new variable, and copy over this value, and then repeat that process for the next year. How do you think I would add these numbers to my list? In this case, I'll add them at the beginning because this is an ordered data structure. So there's 2018, and at the beginning is 2017. Now I want you to think for a moment, how does the code to sum up these values change for my many individual variables and for my list? **[pause for thought]** In the first case, the code changes because I have to add income_2017 and income_2018 to this calculation. You can see this has become a really long line of code. If I hit shift enter, now I get 5.3 million, which is great. In the second case, my code actually doesn't change at all. `sum(incomes)` is going to sum up all those incomes, no matter how large the list is. When I hit shift enter, you can see that these values match.
🎙️ TH	So you've seen that lists are extremely valuable when you have a collection of values that represent the same thing, like incomes, and they have an order to them. That value comes from unlocking all of these cool, useful functions like `sum`, that allow you to perform operations on many values with very little code. Follow me to the next video to learn more about the tasks you can perform on lists.

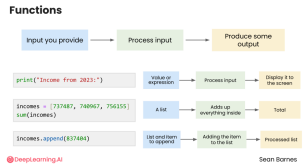
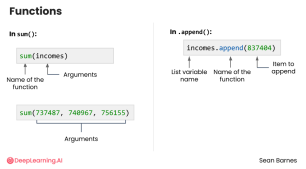
L2V5 – List operations

Visual	Script
🎙️ TH	So you saw lists in action a moment ago. Let's formalize this knowledge and explore more of what lists are capable of.
	As a reminder, lists are **[CLICK]** a type of sequence, meaning **[CLICK]** a collection of values that has an order. **[CLICK]** Let's break down what's happening in this line of code: `incomes = [737487, 740967, 756155, 763415, 830696]`, focusing on the right side of the equal sign for a moment. **[CLICK]** The brackets indicate a list, and **[CLICK]** inside you have individual values separated by commas. In order to use this value later, you'll want to give it a name. So that's what **[CLICK]** `incomes =` means. You create **[CLICK]** a new box,**[CLICK]** label it incomes, and **[CLICK]** store this list inside. Lists often have a **[CLICK]** plural variable name, because they represent a **[CLICK]** collection of values, rather than just **[CLICK]** one value, like the variable `income_2023`. Sometimes these values inside a list are also called **[CLICK]** items. Let's take this over to a notebook to see some more list operations you'll commonly use.
🔗Screencast 🔗Notebook	Okay, so we have three new tasks from the library. We want to sum up the incomes from 2019 and 2021. We want to update the 2023 income. And we want to add a new projected 2024 income to the end of our list of values. Let's get started. Here is our `incomes` list, which matches this table right here. And I added a comment with the year for each of these values. Can you remember how to sum up all the values in this list? **[pause for thought]** Well, we can print the sum of the incomes list, the result that shows is 3.8 million, which you may recognize. Now, let's say you want to access one value in the list. This is something we're going to need in order to add up 2019 and 2021. We don't want to add **all** of these incomes, just the first and third ones. So, in order to access the first value in a list, you're going to start with the name of the list, `incomes`, open bracket, and the position, or index, that you want to access. Now, this is a weird thing, but in Python, and in the vast majority of programming languages, the first item in a list is at position zero or at index zero.

	Close brackets. As a human, this probably feels weird to you because we just don't start counting from zero... no one has a zeroth birthday. But because of the way computers are built, zero is the first index. I went ahead and added the list index to this table to help visualize this concept. So the first value for 2019 is at index zero, the second value is at index one, and so on. What do you notice about the last index? **[pause for thought]** Something interesting about the last index is that it's always 1 less than the number of items in the list. That's because we start counting at 0 instead of 1. Let's hit shift enter and print this out. The first value is 737,487, that looks right. If I were to do index 1 instead, I would get the second item in our list, which is around 740,000. Switching back to the first value at index 0, can you think of how to sum up 2019 and 2021? I'm going to do `incomes`, open bracket, 0, close bracket. We already accessed that one a moment ago. Plus `incomes`, open bracket, 2, close bracket. And now if I hit shift enter, I have about 1.49 million. That checks out. So, we've completed the first task in our list. Let's go ahead and update the 2023 income to 830,000. Let's print out the 2023 income first before we update it. In this case, 2023 is at index 4. So we can print `incomes[4]`. One thing that's interesting about `incomes[4]` is it acts like a variable name. So I can go ahead and update that value using the equal sign just as I would any other variable. Let's say I want to change the value stored in `incomes[4]`. I'm going to write a command that says `incomes[4]` equals 830,000. And let's go ahead and print it out again. We'll see if the value stored in `incomes[4]` did in fact change. I'm going to hit shift enter, and we see that the 2023 income before I updated it was 830,696. And afterwards, it was just 830,000 even. Cool. We've completed that task! Now we want to add a projected 2024 income. This is an interesting task. Let me copy this value first. Now think about what needs to happen here. My list is going to go from 5 items to 6, and I want to add this value to the end. Before we do any adding, I'm going to go ahead and just print out the list as it is now. Let's actually see what `print(incomes)` does. It just prints out the whole list. Let me clear this. Now, how am I going to add this new value? I'm going to take `incomes`, this is our list, and I'm going to use dot append, open parentheses, close parentheses. Then, I'm going to put the item I want to add inside these parentheses. I'm going to paste in that value that I already copied. This dot is a new piece of code. This is a function, similar to sum. What's happening is it's

	taking `incomes`, you're going to append an item to it, and you're appending the value of 837,404. I'm just going to comment these lines out for a second. Here's a reminder of what our list looks like now. What do you expect the list to look like after I run this code cell? **[pause for thought]** I'm going to hit Shift Enter, and now we see that we have all the data we had before, plus this new item that represents the 2024 income at the end of the list.
	Let's quickly recap what you just saw. I know that demo moved quickly. Remember that you can rewatch these demos any time, and slow down or speed up the video based on what feels best for you. First, lists are what programmers call **[CLICK]** zero-indexed, **[CLICK]** the index of each item starts with zero rather than one, which is unusual for us as people and takes some getting used to.**[CLICK]** The index of the last item in a list is the length of the list minus one. You saw that you can **[CLICK]** access items using this code – the **[CLICK]** name of the list, a set of brackets, and inside the brackets is the **[CLICK]** index you'd like to access. You also saw how **[CLICK]** to update an item. So in this code, **[CLICK]** incomes[4] acts like a variable name. And what I mean by "acts like a variable name" is that incomes[4] is a **[CLICK]** name for the fifth item in the incomes list, and no matter what value incomes[4] contains, it will always be called incomes[4]. So you can enter a command of incomes[4] = **[CLICK]** some new value, and the value inside the list will be updated. Finally, you saw how to **[CLICK]** append an item to the list. This used a new dot character. **[CLICK]** Dot allows you to perform an action, in this case the append function, on some data, like a list. So this snippet of code, `incomes.append(837404)` added a new item, 837404 to the end of the list. You'll see this dot many times throughout this course and when working as a data analyst, so it's worth taking some time to practice in the upcoming reading item.
🎙️ TH	Fantastic work with lists! Lists are incredibly flexible and I find them really fun to work with. Before we wrap up this section, let's take a look at functions. You've seen a few functions so far: print, sum, and append. Let's expand your repertoire. I'll see you in the next video!

L2V6 – Taking action: calling functions

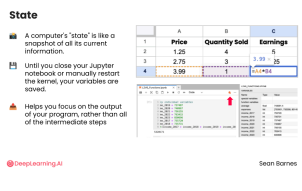
Visual	Script
🎙️ TH	Once you have some data to work with in variables and lists, you'll want to take action using that data! In Python, functions take action on data or take action in the world. Let's check them out.
🖼️ Slides 	You already learned earlier in this lesson that computer programs essentially take some input, process it, and produce some output. Functions are like mini computer programs, they take some **[CLICK]** input that you provide, **[CLICK]** process it, and **[CLICK]** produce some output. • **[CLICK]** The print function takes whatever you want to print–which can be a **[CLICK]** value or an expression–**[CLICK]** processes it, and **[CLICK]** displays it to the screen. • The **[CLICK]** sum function takes **[CLICK]** a list as an input, **[CLICK]** adds up everything inside it, and **[CLICK]** outputs that total. • **[CLICK]** Append is a bit different; there are actually *two* inputs. Can you spot them? **[pause for thought]** The two inputs are **[CLICK]** the list and the item you want to append. Then, it **[CLICK]** processes these inputs by appending the item to the list, and then it **[CLICK]** returns the new list with the appended value.
	Let's break down the code for two of these functions a bit further: 1. **[CLICK]** In sum, you start with the **[CLICK]** name of the function, **[CLICK]** open parentheses, **[CLICK]** enter the inputs, and **[CLICK]** close parentheses. You can also **[CLICK]** add multiple individual numbers if you separate them by commas. The inputs to a function are called **[CLICK]** *arguments*. This term comes from math, and if you find it distracting, you can think of these as just **[CLICK]** inputs, but the formal term is argument. 2. **[CLICK]** Append works a bit differently. On the left you have a **[CLICK]** list variable name. **[CLICK]** Then a dot. **[CLICK]** Then the name of the function, **[CLICK]** open parentheses, **[CLICK]** then the item you want to append. **[CLICK]** Then close parentheses.**[CLICK]** Why is append different? As you can see, one of the inputs, the list, is not inside the parenthesis. It actually comes before the function, but the dot tells the function to also include the list as an input. You'll often see this format when working with lists and other collections. Now that you've seen the code for a few functions up close, let's take a look at some more functions in action!
🔗Screencast 🔗Notebook	Okay, we're back with the Plainville Public Library. They have a couple more questions for us. Don't worry, we're almost done with the Plainville Public

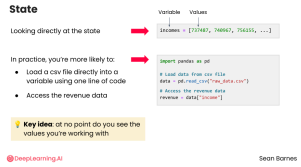
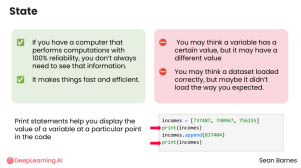
	Library 🙂 They've shared the operating expenses for the past several years, which includes the year they were incurred and the index that they will be in our list, which you can see down here in the code. They would like us to calculate the maximum expenses for all the years, the minimum in just the first three years, and the average of the expenses for all the years. Quick question for you, based on the indices in the right hand column, how many items are in this list? **[pause for thought]** Well, remember that the last index is always one minus the length of the list. So in this case, there are 10 years of data. Here we have our list in Python plus a comment showing the years that they correspond to. And first we'll calculate the maximum expense across all years. Now it may surprise you how straightforward this is. I'm going to enter the print command, and open parentheses. I'm going to use a new function called max, and open another set of parentheses. What am I taking the max of? The list of `expenses`. I'm going to close the parentheses twice; the first set of parentheses corresponds to the max function, and second set ends the inputs for the print function. You can see this highlighting helps you remember which parentheses go with which function. Next, I'll hit shift enter. The maximum value for all years is 801,000. If I go back up, that was in the year 2016. Now you may think, hey, I can just look at this myself faster than writing the code. What's cool about lists and the max function is that no matter how long the list of expenses is, whether it's hundreds, thousands, or millions of items, max will find the maximum value quite efficiently. For a list of one million items, that might take Python abouuuuut a half a second to find the max. So I'm guessing that's faster than you! All right, let's calculate the minimum value for 2014, 15, and 16. Similarly, I'm going to print so we can see the value at the end, and I'm going to use a new function called min. You use min the same way you use max and sum. Open parentheses. Now, I don't want to find the minimum across all of the expenses. I just want to consider the first three years. Can you remember how to access the value for 2014 from the list of expenses? **[pause for thought]** We're going to start with the name of the list, expenses, open bracket, because I want to access an item and I'm going to enter the index zero. That's the value associated with the year 2014. Close brackets. Now I'm going to add a comma, and I'm going to add one more item. `expenses[1]`. And the one more item, `expenses[2]`. These are the expenses corresponding to years 2014, 2015, and 2016.

	Now, I'll close the parentheses for the min function and the print function, since I'm done with all of my inputs. This is the first time you've seen a function with multiple arguments. In this case, I'm finding the minimum between three numbers. The type of expenses is a list, but the type of `expenses[0]` is an integer; just a single number. At a glance, what do you expect to be the output when I run this code block? **[pause for thought]** I'm going to hit shift, enter, and there's our minimum value, 732,000. Nice! Sum and max work the same way. They can find the sum or max using several different integers or floats as input. Let's try something a little more complicated: calculating the average expenses across all of these years. Remember the average calculation involves adding up all of these expenses. You may remember how to do that. And then dividing by the number of expenses. I'm going to write the steps in comments so I remember what I need to do. Add up the expenses, divide by number of expenses, and then print the result. Let's start with the first step, adding the expenses. I'm going to use the sum function, `sum(expenses)`, and I want to save this value for later because I'm going to use it in another calculation in a second. I'm going to assign this result to the variable `total`. In my second step, I'm going to divide the total by the number of expenses. So, total divided by... How do you get the number of items, or the length of a list? You can use the `len` function. `len` stands for length. And it's just going to count the number of items in the expenses list, which you guessed earlier was 10. Essentially, what this is going to do is calculate the total divided by 10, and I'm going to assign this value to a new variable called 'average'. So to recap, I calculated the total by summing up all the expenses. Then I divided that total value by the length of expenses, or in other words, the number of expenses in the list. Then, I saved that value in the variable average. Now, what should I print out here? I'm going to print average. My last question for you is, what do you expect to print out when I run this code cell? **[pause for thought]**

	You may not be able to calculate this in your head, but you should be able to give an educated guess. I'm expecting a value that is somewhere in the low 700,000s, based on the values in the list. I hit Shift Enter, and I get a value around 743,000, so that kind of checks out. That value seems to fall somewhere in the middle of the listed expenses.
🎙️ TH	You've learned so much in this lesson! From types to expressions, variables, lists, and functions, you now have many of the core tools you'll use daily as a data analyst. In the last two videos of this lesson, you'll explore ways to interact with and understand what's happening inside the computer as you code. See you in the next video!

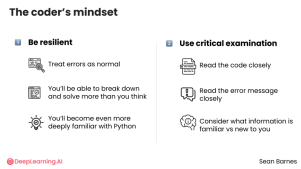
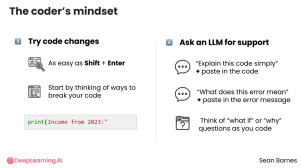
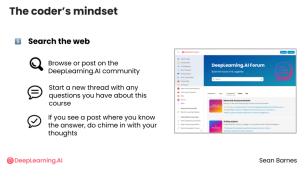
L2V7 – State

Visual	Script
🎙️ TH	As your computer runs each line and each cell of Python code, it keeps track of what effects they've caused. Specifically, it remembers the variables you've created and their values, allowing you to use and modify them later on in your code.
🔗Screencast **Same notebook as prior lesson**	Let me show you something cool. In this notebook, if I right click and say Open Variable Inspector, I get this new window that shows all of the different variables that I have created, as well as their type, their content (or their value), and some other information. This is kind of like a snapshot of the current status of the notebook at this moment. I'm going to drag this window over to the right side so that we can look at the code and the variable inspector together. So, when I ran this code `total = sum(expenses)`, which you saw in an earlier video, that created a new row in my variable inspector `total`, and the current value is now 7.4 million. These variables go back all the way to the beginning of this lesson, because, if I scroll up, there's actually a lot of code here so far. So all of these variables are saved in the memory of the computer, with whatever these particular values are, and that is the current state of the computer. State is a really important concept in computing, because it tells you – where am I at in my computation? What variables have I created? Let's take a look at how this works.
	A computer's "state" is like a snapshot of all its current information. **[CLICK]** Until you close your Jupyter notebook or manually restart the kernel, your variables are saved. **[CLICK]** In a spreadsheet, you're looking at the state directly, because all of the

	data and calculations are displayed in the cells. **[CLICK]** In your Jupyter notebook however, you aren't looking at the state. **[CLICK]** Unless you open the variable inspector, the variables and their values aren't shown to you. This feature **[CLICK]** helps you focus on the output of your program, rather than all of the intermediate steps.
	Here's an example. In the code you saw earlier, one of the lines said something like incomes = and then a bunch of actual numbers. So here you're looking directly at the state – the variables and their values. There's no mystery. In practice, you're more likely to load a csv file directly into a variable like `data` using one line of code. Then, you can access the revenue data using something like `data["income"]`. Don't worry about the code for now, the key idea is just that at no point do you see the values you're working with. You can calculate the max, min, and average without ever seeing them.
	This hidden state gets to the heart of Python's ability to scale. **[CLICKx2]**If you have a computer that performs computations with 100% reliability, you don't always need to see that information, as you would in a spreadsheet. **[CLICK]**It makes things fast and efficient. At the same time, it's a common source of problems. **[CLICKx2]** You may think a variable has a certain value, but it may have a different value. **[CLICK]** You may think a dataset loaded correctly, but maybe it didn't load the way you expected. It's up to you to inquire about the hidden state of your program, typically using print statements. **[CLICK]** Print statements help you display the value of a variable at a particular point in the code.
🎙️ TH	Try not to make assumptions about your program's state. And when your code isn't matching your expectations, try using print statements or the variable inspector to see what values you're actually working with. Follow me to the final video of this lesson, which is all about the coder's mindset of resilience and experimentation. I'll see you there!

L2V8 – The coder's mindset

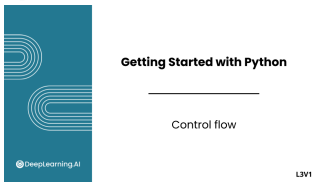
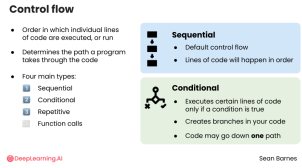
Visual	**Script**
🎙️ TH	I'm guessing that during the practice exercises so far in this course, you've run into errors, or code not working properly. A huge part of coding is encountering things you don't yet understand, like

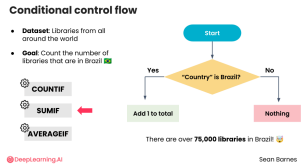
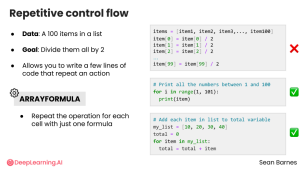
	error messages and unfamiliar code. So many times when I've been coding, I see an error and I just have no idea what it means. How should you approach these moments?
	When you encounter an error or something new that you don't fully understand yet, I want you to try these strategies. • **[CLICK]** First, be resilient. Errors are a part of coding. When you go to write an essay, you don't sit down and write the essay from the first word to the last without using backspace. So **[CLICK]** treat errors as a normal part of the process! **[CLICK]** You'll be able to break down and solve more than you think, and by doing so **[CLICK]** you'll become even more deeply familiar with Python. • Second, **[CLICK]** use critical examination. Don't be intimidated by the strange language of computer-speak. **[CLICK]** Read the code closely, **[CLICK]** read the error message closely, and **[CLICK]** consider what information is familiar vs new to you.
	• Next, **[CLICK]** try code changes. One of the beautiful things about coding is that testing your code is virtually free... it's **[CLICK]** as easy as shift enter. You can **[CLICK]** start by thinking of ways to break your code, like "**[CLICK]** what would happen if I **[CLICK]** left off a quotation mark or **[CLICK]** parentheses", then test it and see if you were right. • Another great option if you need support is to **[CLICK]** ask an LLM. Even a short prompt can be highly effective, like **[CLICK]** "Explain this code in simple terms" and paste in the code, or **[CLICK]** "What does this error mean" and paste in the error message. LLMs are excellent coding buddies because they've been trained on a lot of code on the internet, and it turns out they're quite familiar with Python. You might **[CLICK]** think of "what if" or "why" questions as you code, and a chatbot is a great place to ask those.
	• If you have more specific questions or want to connect with others about your code, I encourage you to search the web. There's a lot of great advice out there. You can also **[CLICK]** browse or post on the DeepLearning.AI community. I encourage you to **[CLICK]** start a new thread with any questions you have about this course, and **[CLICK]** if you see a post where you know the answer, please chime in with your thoughts.
🔗Screencast 🔗Notebook	Here's an example of some code you might be working with. I'm going to hit Shift Enter, and I get an error. Right off the bat, you might look at this and say, wow, there's a lot going on. First, let's talk about being resilient. It's all about your mindset. Errors are super common, so you should try not to get too frustrated when things don't work the way you expected. Let's take a closer look at this. Next, I'm going to use critical examination. This error shows me the same code

	as I had written, and a little carat showing where it thought the problem was. And it says I have a "syntax error incomplete input." I might not know exactly what that means, but based on this information, I can then make some code changes. Now, I know that these quotes match, but I see that there is a missing parenthesis. One thing I could try is removing the parenthesis, and if I hit shift enter. Now I see "missing parenthesis. Did you mean print open parenthesis, close parenthesis. So there's no harm in trying, even though this was the incorrect solution; now I got even more information just from having tried something. So I'm going to go ahead and add both parentheses in here, and shift enter, and now that printed out as I expected.
TH	You're already practicing the coder's mindset: resilience, critical examination, trying changes, and then using an LLM or a coding buddy if you want to chat things through. Once you've completed the practice lab and practice assessment for this lesson, follow me to the next section to learn more about control flow: how to repeat code and create branching code in your programs. See you there!

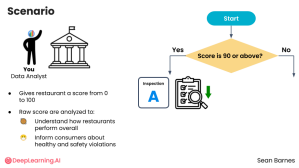
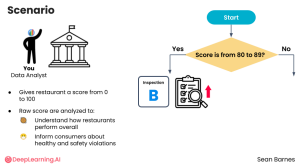
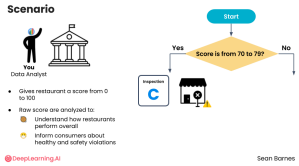
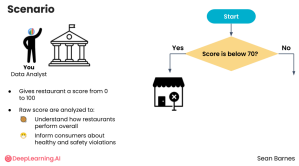
Control flow

L3V1 – Control flow

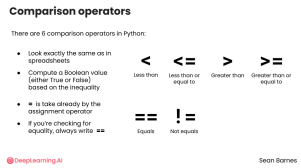
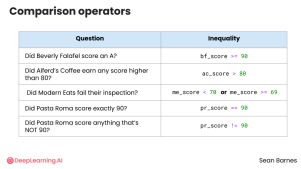
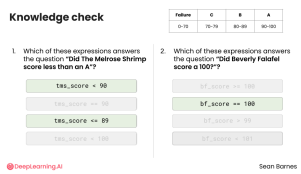
Visual	Script
TH 	In the programs you've written so far, each line in a cell was executed from top to bottom in order. However, we often need our programs to make decisions, repeat tasks, or skip certain sections based on conditions. This process is called control flow.
	Control flow is the **[CLICK]** order in which individual lines of code are executed, or run. It **[CLICK]** determines the path a program takes through the code. There are **[CLICK]** four main types of control flow: **[CLICK]** sequential, **[CLICK]** conditional, **[CLICK]** repetitive, and **[CLICK]** function calls. You'll learn about function calls in the next module, so let's focus on the **[CLICK]** first three for now. You've already seen **[CLICK]** sequential control flow in the code you've created so far. It's the **[CLICK]** default way things work. Unless you introduce some of

	the control structures you'll learn about in this lesson, your **[CLICK]** lines of code will execute in order. Next up is **[CLICK]** conditional control flow. Conditional flow **[CLICK]** executes certain lines of code only if a specified condition is true. It **[CLICK]** creates branching paths in your code, kind of like a fork in the road, where your **[CLICK]** code may go down one path or the other, but not both at the same time.
Use a conditionals diagram for libraries example 🔗Source 	Let's draw out a real world example. Say you're working with a **[CLICK]** dataset of libraries from all over the world, and you'd like to **[CLICK]** count the number of libraries that are in Brazil. For the first library, you would look at its country and then take action depending on that value. **[CLICK]** If the value in "Country" is "Brazil", then you **[CLICK]** add one to your total. If the value is **[CLICK]** anything else, then you **[CLICK]** do nothing. By the way, **[CLICK]** there are over 75 thousand libraries in Brazil! Remember that you have already encountered conditionals in spreadsheets. Functions like **[CLICK]** COUNTIF, **[CLICK]** SUMIF, and **[CLICK]** AVERAGEIF, could handle different conditions. For example, **[CLICK]** SUMIF could look at a value, and add it to the total based on a particular condition, which is exactly the task happening in this diagram.
	Next up, you have repetitive control flow. If you have **[CLICK]** 100 items in a list and you need to **[CLICK]** divide them all by two, **[CLICK]** you don't want to have to write 100 lines of code, one for each item in the list. Repetitive control flow **[CLICK]** allows you to write a few lines of code that repeat an action. Examples could be **[CLICK]** "print all the numbers between 1 and 100" or **[CLICK]** "Add each item in this list to my total variable." To make a spreadsheet comparison, repetitive control flow is similar to what happens with an **[CLICK]** ARRAYFORMULA function. You have some operations, like text processing, and you want to perform it on many cells, without having to manually drag the handle down. ARRAYFORMULA will **[CLICK]** repeat the operation for each cell with just one formula.
🎙TH	Control flow gives you a ton of flexibility when designing programs. Follow me to the next video to learn the first building block of conditionals in Python: comparisons.

L3V2 – Comparison

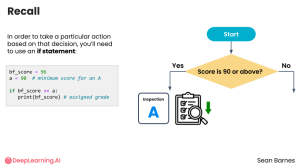
Visual	Script
TH 	You learned in Data Analytics Foundations that data is any information that can be used to make a decision. So how can you make decisions based on data in Python?
Same conditional type diagram Also for the A B and C signs it would be cool if we had something looking like the actual sign in a window (icon is fine) to kind of emphasize the real world aspect of it    	Say you're working with a **[CLICK]** government agency that performs **[CLICK]** restaurant inspections. The inspectors take a tour of the restaurant and **[CLICK]** give it a score from 0 to 100 based on its sanitation. These **[CLICK]** raw scores are then analyzed to **[CLICK]** better understand how restaurants are performing overall, and to **[CLICK]** inform consumers about health and safety violations. **[CLICK]** If a score is 90 or above, it's considered an A. The restaurant can display the **[CLICK]** A grade publicly, and it's typically **[CLICK]** inspected less frequently. **[CLICK-Next slide]** A score from 80 to 89 is considered a B. The restaurant must **[CLICK]** display the B grade, and it may be **[CLICK]** inspected more frequently. **[CLICK-Next slide]** A score at 70-79 is considered a C. The restaurant must **[CLICK]** display the C and will need to take immediate corrective action or **[CLICK]** face closure. Anything below 70 is considered a failure, which **[CLICK]** leads to immediate closure. You can see how the different scores branch out into different courses of action.

🔗**Screencast** 🔗**Spreadsheet** 🔗**Dataset**	Say you're given a dataset like this one, adapted from one released by the County of Los Angeles, California. You have a column of scores, but they need to be converted to ratings in order to send the correct signage and schedule necessary inspections or closures. This dataset has 67,000 rows, each one representing an inspection, so even though you could do this manually, you certainly wouldn't want to. Let's use a random sample of 5 rows to see how you would go about converting these scores to ratings.
🔗**Screencast**	Let's start by making some comparisons. Here are five different restaurants and the scores that they received. The first thing I'm going to do is create some variables to store these values. I'm going to use the initials for the restaurant, like bf underscore score equals 96. Pr score equals 91. And so on. Now I can use these restaurant scores in my code. I'm going to create a variable to store the cutoff score for an A. A equals 90. This is the minimum score for an A. The first thing I might ask is, is the score for Beverly Falafel an A? To do that, I'm going to print out the result. BF score greater than or equal to A. What do you think will be printed? **[pause for thought]** Well, the value of BF score is 96. The value of A is 90, and 96 is in fact greater than 90, so when I hit Shift Enter, I get the value true. Now I can do the same for other scores. Let's check whether the TMS score is greater than or equal to A. Is the Melrose shrimp an A? If I hit Shift Enter, the answer is false. The Melrose shrimp has a score of 79. That's actually a C. Greater than or equal to is just one of many comparison operators. These comparison operators take in two values, one on the left and one on the right, and they answer a question about those values. Is BF score greater than or equal to A? I can also ask something like, is TMS score less than 80? Just as an example. I can print TMS score less than 80. And if I hit shift enter, the answer is true. When you're asking a question like, is tms score less than 80, what are the possible answers? Well, there's only two. Either the answer is true, tms score is

	less than 80, or false, it's not. There's no in between. You may notice that there's only two possible outputs of these comparison expressions, True or False. This type of value, the output, is called a boolean. We'll talk about it more in a moment, but a boolean just means either true or false. and it's just another data type in Python, just like int, float, and list. Let's take a closer look at comparison and booleans and the different comparison operators you have at your disposal.
	There are 6 comparison operators in Python. First you have **[CLICK]**less than, **[CLICK]** less than or equal to, **[CLICK]** greater than, and **[CLICK]** greater than or equal to. These **[CLICK]** look exactly the same as in spreadsheets and **[CLICK]** compute a Boolean value (either True or False) based on the inequality. Next you have **[CLICK]** equals, which is written with two equals signs, and **[CLICK]** not equals which is written as exclamation point equals. Why do you think the equality operator has two equals signs? **[pause for thought]** That's because the**[CLICK]** single equals is taken already by the assignment operator, the one used to assign a value to a variable. It's important to remember, **[CLICK]** if you're checking for equality, always write two equals signs.
	Here are some questions you might have, and the inequality that would answer each one. • **[CLICK]** Did Beverly Falafel score an A? You already saw this could be answered with **[CLICK]** `bf score >= 90`. • **[CLICK]** Did Alferd's Coffee earn a score higher than 80? That could be calculated using the code **[CLICK]** `ac score > 80`. • **[CLICK]** Did Modern Eats fail their inspection? In that case, you could use the code **[CLICK]** `m e score < 70`. Can you think of another way to answer this same question? **[pause for thought]** That would be **[CLICK]** `me score <= 69`. This inequality answers the exact same question, assuming the scores are only integer values.. • **[CLICK]** Did Pasta Roma score exactly 90? You could use the code **[CLICK]** `pr score equals equals 90`. • And **[CLICK]** did Pasta Roma score anything that's NOT 90? That would be **[CLICK]** `pr score != 90`.
	Let's put your knowledge to the test with two questions! • **[CLICK]** Which of these expressions answers the question "Did The Melrose Shrimp score less than an A"? **[pause for thought]** That would be **[CLICK]** `tms_score < 90`, or `tms_score <= 89`! Both of these

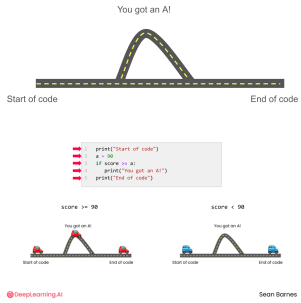
	expressions check whether the value for Melrose Shrimp is anything less than 90, the score for an A. • **[CLICK]** What about the question, “Did Beverly Falafel score a 100?” **[pause for thought]** The correct answer is **[CLICK]** `bf_score == 100`.
🎙️ TH	Excellent work with comparison and booleans! These operators allow you to make decisions in your code using conditionals. Follow me to the next video to see how.

L3V3 – Branching code: if

Visual	Script
🎙️ TH 	Now that you’ve practiced with comparisons, let’s see how to use those comparisons to make decisions in your code.
Similar flowchart type diagram 	Recall that your task is to assign a grade based on the restaurant’s score. **[CLICK]** You already saw the code that makes that comparison for an A: a variable containing the score greater than or equal to 90. **[CLICK]** In order to take a particular action based on that decision, you’ll need to use an if statement. For example, your first step might be to **[CLICK]** simply print out the assigned grade. Let’s take a look at the code.
🔗Screencast 🔗Notebook	So, our task is to check if each restaurant has received an A rating. Remember, our ultimate goal is to scale this operation to all 67 thousand rows. Let's build towards that task step by step. You already saw in the previous video that this comparison will check whether the variable bf_score is greater than or equal to A, which is 90, the minimum score for an A. I'm going to copy that code, and I want to use that comparison to take some kind of action. To do that, I'm going to use an IF statement, so I’m going to type IF, and then I’ll paste in the comparison that you saw earlier, bf score greater than or equal to a, colon, and now you'll see that when I hit enter, My code is indented. The colon and indentation indicate that all of this code belongs together as part of the IF statement. I'm going to start a new block of code, and this block of code is going to run only if bf score is greater than or equal to A. Let's enter a print statement here, which will say “you got an A!” if the condition is True. Remember that BF score

	is 96, and A is 90. What do you think will happen when we run this code? **[pause for thought]** I'm going to hit shift enter, and this line of code does run. You got an A. Now, let's do the same for Pasta Roma, which has a score of 91. Remember, the variable for the Pasta Roma score was PR score, so I'll just copy that. If PR score is greater than or equal to A, I want to print, you got an A. What do you think will happen when I run this code? **[pause for thought]** If I hit Shift Enter, you got an A. Now, remember the Melrose Shrimp? They were having a tough time. Let's check if the Melrose Shrimp score is an A. Again, I want to print, you got an A. Only if the Melrose Shrimp did, in fact, get an A. What do you think will happen when I run this code? This may be a bit trickier. **[pause for thought]** If I hit Shift Enter, well, nothing happened. That's because the score for the Melrose Shrimp is a 79, which is not an A. That is, in fact, a C. What happened in this cell is that first the code checked if The Melrose Shrimp got an A. Is the score 79 greater than or equal to 90? This comparison was false, so the line of code that says to print “you got an A”, was never executed. The program didn't run that line. Let's do the same for modern eats. If I hit shift enter, you got an A. Modern eats had a 93. What about Alferd's coffee with an 86? If I hit Shift Enter, again, nothing happens. We checked if AC score is greater than or equal to A. 86 is not greater than 90, so this comparison is false, and the print statement did not run. Let's recap this process using our flowcharts.

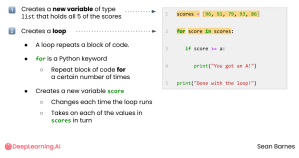
Please add line numbers to the code
 	
Let’s show the lines executed, path, and output for each option
 	
It would be great if we could illustrate the idea of branching paths here, and show the paths side by side,
	Python
	`print("Start of code")`
	`a = 90`
	`if score >= a:`
	`    print("You got an A!")`
	`print("End of code")`
	
	Here are a few lines of code that are very similar to the notebook you just saw. Let me ask you, how many paths are there through this code, how many

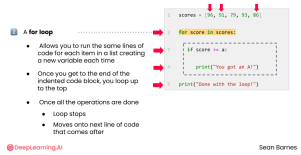
quick mockup, it does not have to be like this: 	different ways could this code execute? **[pause for thought]** There are two different ways! It depends on the value of score. **[CLICK]** If score is in fact greater than or equal to 90, then lines 1 2 3 4 and 5 will all execute, and the program will print **[CLICK]** "Start of code", **[CLICK x3]** "You got an A", and **[CLICK]** "End of code". **[CLICK]** If score is less than 90, then the computer will take a different path through the code. Only lines 1 2 3 and 5 will execute, and the program will print **[CLICK]** "Start of code", and **[CLICK x3]** "End of code".
🎙TH	Branching code is fundamental to programming. In data analytics, not all data is relevant to the question you're trying to answer, and different types of data need to be processed differently.

L3V4 – Repeating actions: for loops

Visual	Script
🎙TH 	You've now seen how to check one score at a time and turn it into a grade. At the moment, that's no better than doing it manually. To realize Python's efficiency, you'll need to repeat actions in your code. Let's take a look at why you might want to do that and how to do it.
🔗Screencast 🔗Notebook	Take a look at these five code blocks you saw in the previous video. What do you notice about all five of them? They look quite similar. They all have if, some kind of variable representing the score of the restaurant, greater than or equal to A, colon, and print you got an A. The only difference between these five code blocks is the name of the score variable. What if there was a way to write this code so that we had one variable name that changed for each of the five restaurants? Let's see exactly how to do that. First, I'm going to copy this code from the previous video. Here, I have all the scores of the restaurants in a list. Next, I'm going to create a **for** loop. This **for** loop is going to look at each one of these values in turn, and print out, you got an A, only if that score is an A. In order to do that, I'm going to write, for score in scores colon. This is really the only new line of code you're going to see in this video. I'm going to hit enter, and you'll notice again that my code is indented. This is a common feature of control flow. You have some code that's indented that you want to run only under certain conditions or for a certain number of times.

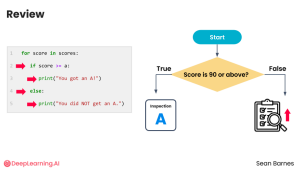
	The first thing I want to show you is what if I just print score. What do you expect to print out once I run this code? **[pause for thought]** I'm going to hit shift enter and you can see that all five scores are printed from the list. What's happening here is that this line of code, print score, is running one time for every score in the list scores. This repetitive loop will run a total of five times, the same as the length of the list. each time that line of code runs, score has a new value. It takes on, one by one, the values in this list scores. The variable name `score` and `scores` help you remember what these different values represent, and you can choose other names if they make more sense to you, like `restaurant_score` and `restaurant_scores`. Suppose I delete the code within this **for** statement, and paste in the code that we had before to check the score of Alferd's Coffee. What do I need to change to make this work with my loop? **[pause for thought]** I'm going to just remove AC underscore. Now, for every score in scores, the code is going to say, if the score is greater than or equal to A, which we know is 90, then it will print "You got an A." Essentially, we're able to run these five code blocks with only these few lines of code. It's much shorter, right? OK, I have one more question for you. What do you expect to print out when I run this code? Hint! How many A's are in the list scores? **[pause for thought]** If I hit shift enter, now I have printed, you got an A three times. Let's step through this code one more time, using our diagrams.

	Python
	`scores = [96, 91, 79, 93, 86]`
	
	`for score in scores:`
	`  if score >= a:`
	`  print("You got an A!")`
	
	`print("Done with the loop!")`
	
	What does each component do? The first line, **[CLICK]** scores equals, **[CLICK]** creates a new variable of type **[CLICK]** list that holds all **[CLICK]** 5 of the scores you saw in earlier videos. **[CLICK]** The line `for score in scores:` creates a loop. **[CLICK]** A loop repeats a block of code. **[CLICK]** The `for` is a Python keyword that indicates you're going to **[CLICK]** repeat that block of code **for** a certain number of times. **[CLICK]** `score in scores` **[CLICKx2]** creates a new

	variable `score` that **[CLICK]** changes each time the loop runs, and it **[CLICKx2]** takes on each of the values in `scores` in turn.
	Stepping back for a moment, **[CLICK]** this line of code – called **[CLICK]** a **for** loop – **[CLICK]** allows you to run the **[CLICK]** same lines of code for each item in a list without creating a new variable each time. It's called a loop because **[CLICK]** once you get to the end of the indented code block **[CLICKx3]**, you loop back up to the top of it. **[CLICK]** Once all the operations are done – **[CLICK]** in this case, once the loop reaches the last item in the list – **[CLICK]** the loop stops, and the computer **[CLICK]** moves on to the next line of code that comes after.
🔗Screencast Same notebook as above	Here's a quick demo of why this is powerful. If I have a new list with 147 restaurants instead of 5, now the operation is a lot more complex. What's cool about loops is that I can take this code, copy it and paste it here. And now, because the name of my list of scores is the same, I can hit shift enter, and get the results for **all** of those restaurants without much additional effort. Wouldn't it be useful if we were able to not just say you got an A, but to respond to more possible cases? For example, if we come to this third restaurant, it didn't get an A. What if we could print something about that as well? Let's take a look at how to create multiple branches in your code.

L3V5 – Branching code: else

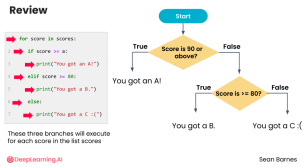
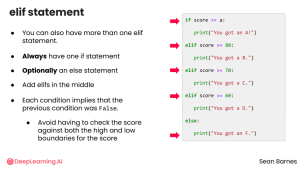
Visual	Script
🎙️ TH 	Once you have loops, you're fully equipped to work with incredibly large data sets with only a few lines of code. Let's introduce multiple code branches into this workflow.
🔗Screencast 🔗Notebook	If you take a look at this code and imagine that it was applied in a spreadsheet, what would that look like? **[pause for thought]** You have a loop that runs once for every score, printing you got an A if that score was greater than or equal to 90. Here's that spreadsheet again, here's the score column, here's where you want to put the grade. Let's start the formula equals if, reference the corresponding value from the score column, greater than or equal to 90, then the value is A. But the "if" function also has another input called value_if_false. So you can create a different branch in your spreadsheet formula. Let's just put in an empty string for a moment. This is just going to display nothing if the condition is false. If I

	hit shift enter, well this score was a B, so the grade cell is empty. I'm going to double click the fill handle to add a value for every row. Let's go back to the notebook. The code executed here essentially generates this same output. Any time a restaurant had a score of 90 or above, the code printed an A. I'm going to scroll down a little bit so you can see. Anytime there was not an A, well, we didn't get anything. That's *kind of* useful, but this doesn't create a complete grade column. What if I wanted to fill this value, if false, with something more meaningful? For example, I could put not an A. It's either an A or it's not an A. Two options. Two possible branches. This one is not an A. I'm going to clear the grade column and use an array formula. Remember that the array formula kind of acts like a loop. I'll apply this formula to the whole D column. And now we have "not an A" displayed every time a restaurant had a score that is not greater than or equal to 90, whether it's a C or a B. How do we do this in code? Remember we had these 147 restaurant scores. I'm going to go ahead and right click, clear cell output, so you don't have to look at that long output. Let's copy over this code. So if score is greater than or equal to A, Let's print, you got an A. Now I'm going to say else, which means in any other case other than the score being greater than or equal to 90. Let's print "you did not get an A" in this case. I'll make sure I close the double quotes and parentheses. What do you think will happen when running this code? **[pause for thought]** Hitting shift enter, now if I scroll back up, I'll see that the first two values were an A, the next one was not an A. That's correct, it was a 79. The next one was an A, and the next one was not an A, followed by a bunch of As. This else statement is so useful for taking some kind of action no matter the comparisons you have already evaluated. So, if some condition is met, you want to go down one branch in your code. Otherwise, else, you want to go down another branch in your code.
	To quickly review this concept using a flowchart, you **[CLICK]** check the condition whether each score is greater than or equal to 90, then **[CLICK]** take a specific action based on that score. If the **[CLICK]** result of checking that condition is the boolean value True, you **[CLICK]** print "You got an A!". This branch is the equivalent of **[CLICK]** sending the restaurant an A sign. **[CLICK]** If the boolean value is False, **[CLICK]** you'll print "You did NOT get an A". That branch is the equivalent of **[CLICK]** signing the restaurant up for more frequent inspections.
TH	Now, two branches are useful, but ultimately you want to have more than that – you want to assign grades A, B, and C, and take action accordingly. Follow

	me to the next video to see how to do that.

L3V6 – Branching code: elif

Visual	Script
🎙️ TH 	Let's put it all together – loops and conditionals – to assign restaurant food safety grades. The last piece of the puzzle is creating more than two branches in your code using control flow.
🔗Screencast 🔗Notebook	This is where you left off in the previous video. Let's finally assign the scores for every restaurant! What's missing here in order to do that? **[pause for thought]** What's missing is the ability to assign a B or a C. I'll keep this code as it is and just copy it. Let's also right click clear cell output because it's just so long. Remember this code has two branches: A or not an A. Now let's do A, B, or C: three branches in the code. Let's add a new branch: Else if, meaning essentially let's check one more thing. If score is greater than or equal to 80, you want to print, you got a b. Maybe this should end with a period. It's not as exciting to get a B. Notice that all three of these branches provide unique pathways for the code to execute. You can only take one of them, and that will be based on how the comparisons work out for a given score. You can kind of think of them like a progressive filter. Any scores that are greater than or equal to 90 follow this branch in the code, and you print "you got an A!" The elif statement is going to check another condition. Is the score greater than or equal to 80? If so, you got a B. Now, let me ask you, how should I change the else print statement? **[pause for thought]** While there's only three possible scores, the third branch in the code is you got a C. I'll write this one with a sad face, since that means the restaurants that meet this condition weren't doing very well. Now, this code has three branches: A, B, and C. Let's run this code with Shift Enter. The results are an A, an A, and a C. Let's take a look back at the list of restaurant scores. Yep, 79 was a C. Followed by an A, followed by a B. Here's an A, here's a B. Let's go back to the spreadsheet as a comparison. In order to do that in a spreadsheet, you would have to use the IFS function. IFS essentially does the exact same thing as our if-elif-else statement in Python. You check one

	condition. If this is true, give it an A. If it's false, check the next condition. If that second condition is true, give it a B. If it's false, check the next condition. This last condition is essentially the else statement. What's interesting here is that we don't really need to check any condition. In Python, we can use the catchall "else". Essentially we can just say, hey, if it's not an A or a B, it's a C.
	Let's model that decision using a flowchart, just to solidify what you've learned. **[CLICK]** The first IF statement checks whether the score is greater than a, or 90. **[CLICK]** If True, then the code will print **[CLICK]** "You got an A!". **[CLICK]** If False, then **[CLICK]** the code will move down this path to check if the second elif statement is True or False. **[CLICK]** If it's True, then the code prints **[CLICK]** "You got a B." If that statement is **[CLICK]** False, then the code **[CLICK]** moves into the Else block, **[CLICK]** printing "You got a C" with a sad face. So, you can see the three possible branches in the code – A, B, or C. Remember that all of this happens inside the **[CLICK]** for loop, so **[CLICK]** one of these three branches will execute for each score in the list scores. If there are 5 items, this chain of decisions happens 5 times. If there are 147 items, it happens 147 times.
	**[CLICK]** You can also have more than one elif statement. You can create as many branches as you need. You **[CLICK]** always have one if statement, then **[CLICK]** optionally an else statement, and if you need more than two branches, you can **[CLICK]** add elifs in the middle... **[CLICK]** one elif, **[CLICK]** two elifs, or as many as you need. In the example you worked with throughout this lesson, you used a score to determine food sanitation ratings from A to C. **[CLICK]** But, if you were working on converting student test scores to A through F letter grades, like the ones used in the US and some other countries, you might need branches for A, B, C, D, and F. That code would look very similar to what you've already seen, but with two extra elifs to compare the score with **[CLICK]** 70, then **[CLICK]** 60. You'd also change the **[CLICK]** print statement for the else condition to represent an F. Notice the descending pattern in the scores. This pattern takes advantage of the sequential nature of IF statements. **[CLICK]** Each condition implies that the previous condition was False. That way, you **[CLICK]** avoid having to check the score against both the high and low boundaries for the grade.
TH	Great work creating branching code! You'll learn more nuances in the next module. For now, follow me to the final video in this lesson, where you'll take an in depth look at execution order. I'll see you there.

L3V7 – Execution order

Visual	Script
🎙️ TH 	Execution order is the sequence in which your program runs individual lines of code. You've seen how control flow allows you to repeat code or take one of several branches in order to make decisions. Execution order can get complex. Let's trace through an example.
🔗 **Screencast** 🔗 **Notebook**	Say you've been asked to calculate the percent of ratings that are As, Bs, and Cs. What information would you need in order to tackle this problem? **[pause for thought]** You'll start first by creating variables for the score cutoffs. So you know that an A is a 90, B is an 80, C is a 70. You're going to need the scores themselves too, so you can start by using the list of five random scores. Stepping back for a moment, what you want to do is look at each score in turn and count up the A's, B's, and C's. So, since you know you want to look at each score in turn, you're going to need a loop. It's a good idea to write comments to keep track of the logic of the code that needs to be written. You want to create a loop to look at each score. And inside that loop, you want to check whether each score has an A, B, or C, and then add 1 to the appropriate tally. When you think about the meaning of this task, it needs to happen inside the loop because it's an operation that happens after you check every score. This code is also going to involve a conditional, because you want to take different paths depending on what the score is. If it's an A, you want to tally it up as an A. If it's a B, you want to tally it up as a B, and so on. Finally, at some point, you'll need to calculate the total number of scores. After that, you can display the percent of all scores that are A's: the number of A's divided by the number of scores times 100. Lastly, you'll need to define variables to store the number of A's, B's, and C's. You want to create these variables above the loop so the loop can use them. I going to create a variable called numAs, and initialize this variable with the value of zero. You haven't done any counting yet, so let's start the count at zero. Same thing for the number of Bs, and the number of Cs. Now, let's go ahead and start the loop with `for score in scores`. Now, you want to check out each score inside the list scores. If the score is greater than or equal to an A, which is code that you've seen before, you want to add 1 to the number of A's. Number of A's plus equals one. The plus equals operator allows you to take whatever value is already inside numAs, and add one to it. It is equivalent to the statement that the number of A's equals the number of A's plus 1.

	Now, there are two other possibilities. Next, check if the score is a B. Elif, score greater than or equal to B. What code do you think I should run if the score is greater than or equal to an 80? **[pause for thought]** In this case, num Bs plus equals one: add one to the number Bs. Then finally, the else statement... in any other case... it's a C, right? Num C's plus equals one. Now we have the same three branches in your code as you had before, but instead of printing out that each restaurant got an A, B, or C, I'm counting up the number of A's, B's, and C's in this list `scores`, and storing the results in the num variables I created. It's good practice to add print statements to track your code. So for example, you can say print adding one to the number of As. If this branch of code executes, this line of code will print what's happening. Same for b's, and same for c's. That just allows you to visualize the path your code is taking. Now, let me ask you, what do you think will be printed when this code runs? **[pause for thought]** Hitting shift enter, first, notice that there are five print statements. That makes sense because this loop runs five times, once for each score. And inside that loop, while there are three different print statements, remember these if, elif, and else paths are exclusive. You can only take one of these paths for each execution of the loop. Each score only counts towards one grade. The first two scores are As, the third one is a C, the next one is an A, and the last one is a B. Now, can you remember how to calculate the total number of scores in this list, or the length of this list? **[pause for thought]** You could say, total equals len scores. There are five of them. And to find the percent of all scores that are A's, you can say the number of A's divided by the total times 100. Remember that num A's starts out as zero, but as your loop runs, it will increase by one every time a score is an A. So, what do you expect to print out now when I hit shift enter? **[pause for thought]** You still have the same print statements as before, plus 60.0 . This is a float representing that 3 out of 5, or 60 percent, of those scores were A's. Let's remove these print statements and rerun the code by hitting Shift Enter. Again, you get 60%, but without the printed statements for each score. The values for the number of A's, B's, and C's are part of the hidden state of the computer, but the percent of A's is calculated just the same. Next, you can add some code to calculate the percent of Bs. Can you think of how to modify this code to calculate the percent of Bs? **[pause for thought]** That code will be num B's, divided by total, times 100. And same for Cs.

	Instead of numAs, it's numCs. I'll delete this empty line. Now, if I hit Shift Enter, this code prints 60 percent are As, 20 percent are Bs, and 20 percent are Cs. You probably could have glanced at this short list and concluded the same quickly. Let's grab this much larger list of scores to replace the variable scores. This loop is going to run 147 times. If you hit shift enter you get, 88 percent of the restaurants got an A, 10 percent got a B, and 1 percent got a C. Let's trace through this code and see how it's working in more detail.
slides – I don't have the script ready yet, but will involve a flow chart of the code at the end of the notebook (the last cell only), tracing through and iterating through a loop. I want to show how the variables change over time.	Let's keep track of the values of each variable. **[CLICK]** Pay attention to the control structures. First, which variables do you expect to stay the same as this code runs? **[pause for thought]** **[CLICK]** A, B, C, and scores will stay the same, since the cutoff scores are constant, and you don't expect to be appending to the list or doing any other operations on it. Meanwhile, **[CLICK]** num A's, **[CLICK]** num b's, and **[CLICK]** num C's will change as the **[CLICK]** scores are tallied. The value for **[CLICK]** score will also change as the loop runs, so you can watch the values of each tally and score. **[CLICK]** Zooming in on the loop, in the first cycle, or iteration, **[CLICK]** score gets the value 96. Then, **[CLICK]** the computer checks if the score is greater than or equal to A. A has the value 90, and **[CLICK]** 96 is greater than 90, so the computer will execute the line **[CLICK]** underneath, **[CLICK]** adding 1 to the variable num A's, which is now 1. The computer then **[CLICK]** skips the elif and else blocks, and returns to the first line of the loop. In the **[CLICK]** next iteration, score gets the value 91. The first thing the computer does is **[CLICK]** check if the value is greater than or equal to A, which is 90, and since that expression is **[CLICK]** True, it executes the **[CLICK]** next line to **[CLICK]** add 1 to the variable num As, which is now 2. **[CLICK]** The elif and else blocks are skipped, and the loop returns to the top. In the next iteration, **[CLICK]** score has the value 79. **[CLICK]** The computer first checks if score is greater than or equal to A, which is 90, but that expression is **[CLICK]** false, so the next line of code is skipped. Then, the computer checks the next condition: **[CLICK]** else if score is greater than or equal to b. Since B is 80, this expression is also **[CLICK]** False, so the computer skips the next line and **[CLICK]** moves to the else block. There's no condition to check in the else, since it's a catchall, so the computer **[CLICK]** executes the line under else, **[CLICK]** adding one to the number of Cs, which is now 1. **[CLICK]** The loop then returns to the top. **[CLICK]** Now score has the value of 93, **[CLICK]** the first condition is True, so the computer **[CLICK]** executes the line underneath and **[CLICK]** adds 1 to

	num A's which is now 3. **[CLICK]** The loop returns to the top. Finally, **[CLICK]** score gets the value 86. **[CLICK]** The first condition is False, since 86 is not greater than or equal to 90. **[CLICK]** The next line is skipped. Then the computer checks else if the score is greater than or equal to 80, which is **[CLICK]** True, so the line underneath the elif is executed, **[CLICK]** adding 1 to num B's which is now 1. Since 86 was the final value in the list, the loop is done, and the **[CLICK]** computer can move on to the next line of code, which involves calculating the length of the list.
🎙 TH	Great work tracing through that code! You just constructed a common coding pattern: tallying up different kinds of items in a list. It's the same operation as SUMIF in a spreadsheet! You've now built up the core skills needed for computer programming in Python: data types, variables, conditionals, and loops. You've covered a lot of ground in just one module, and I'm excited for you to try out the upcoming assessment and graded lab. In the lab, you'll work with data from a local shop to calculate sales for different types of products. Once you're done with the assessment and lab, join me in the next module to explore the data structures and techniques that will allow you to work with not just one list of data, but entire spreadsheets in your code. Great work so far, and I'll see you there!