Basic Computing Actions

You are most likely watching this video because you are in a class where your professor is expecting you to use computing for some of the course activities. In today's educational environment, it is possible that you may work in several different programming languages as you go through your studies. For example, in an economics course, you might use the R language. But in a subsequent sociology course, you might use Python. In physics you might use MATLAB or use JavaScript. You might even find yourself programming in multiple languages during the same semester. You will have a much easier time shifting across these different languages and course requirements if you have some understanding of the basic concepts of computing, of the basic actions a computer can carry out. The goal of the video series you will be watching is to explain basic computing concepts and actions, and supplement the computational work your professor is asking you to do in your course. There are a number of benefits of understanding

computing concepts. It will become easier for you to learn new languages in the future. It will be easier for you to switch between programming languages. It will be easier for you to follow example code and adapt it to solve new problems. It will help you understand problems, solutions that are provided to you, because you will have better ability to step through those solutions on your own. It will help you develop problem solutions of your own. And it will help you figure out why code isn't working because we all sometimes write code that does not do what we want or what we expect it to do. What are these basic actions? They are relatively straightforward and we see examples of them even in day to day life. First, there are sequential operations that are executed in order. For example, think about the basic process for writing a paper for a course. Minimally we hope you go through these steps. You decide on the paper topic. You locate related resource materials. You read the materials and take notes. You make an outline for the paper. You write the paper, you edit the paper, and you turn in the paper. The next basic operation is decisions that determine whether a particular set of operations will be carried out or not. These make it possible for the computer to choose between several operations, or between several groups of operations. You make decisions all the time. And we can make the computer carry out that same sort of process. So you can see here two examples. If it is raining, then take umbrella to class. You make that decision when you look out the window in the morning. Or another decision, if class is at 8:30, then get to breakfast by 7:45. Otherwise, meaning your class isn't at 8:30 and probably is later, then you want to get to breakfast by 9:00. The next basic operation or action is iteration, which refers to repeating a series of operations. Repetition can be based on a fixed count or based on some condition being true. For example, when spreading frosting on a cake, you basically carry out a repetitive process. While cake not fully frosted, so you look at the cake, you see if there's any spots that don't have frosting yet, if the cake's not fully frosted, you will get more frosting on your spatula and spread it on an uncovered section of the cake. And so you can imagine, if you put just a little bit on the cake each time, this process will repeat a pretty large number of times. If you put more frosting on each time you spread some, it won't repeat as many times. But the number of times you would go through this process is not predetermined. It is determined by how long it takes to get the cake fully frosted. The next basic concept is creating and naming functions. This is when we group together and name a series of operations so we can later on refer to the whole group as a single step and use it in the solution to a larger problem. For example, we could take our cake frosting iterative process, name

it simply FrostCake, and then refer to that by name every time you want to frost a cake. All programming languages have ways of carrying out these different problem-solving actions or techniques. Just like natural languages that we speak, each programming language has its own syntax or grammar, and they all have a way to say the same things. For example, these two code blocks will do exactly the same iterative process and generate the same output. But one is a Python version and the other is a Java version. Once you are familiar with the basic computing actions and you know how to carry them out in one language, it becomes much easier to learn subsequent languages in the future. It also becomes easier to think about problem solutions in ways that can be expressed in computer code and executed by a computer. Hopefully, this overview and the material in the subsequent videos will help you with the computing work that you will be doing in your courses.

Assignments and Naming

This is the first video and we're going to be introducing the concept of variables and the concept of assignment. So I want to start with a real-world problem. Let's say we have three buckets, each partially filled with water. And we want to combine them together into fourth bucket to see how much water there is total. It's possible to describe the process of combining the contents of the three buckets without actually knowing how much water is in each bucket. For example, I can take each bucket in turn and dump the contents into the bucket that's initially empty. So first we dumped the first bucket into the last bucket, then we dumped the second bucket into the last bucket. Finally, we dumped the third bucket into the last bucket. Now this process is called an algorithm. And I didn't have to describe any numbers or specifics. I just had to describe the process by which we combine the contents of the buckets together. Now, to describe the process precisely, I have to be able to specify which bucket is which. If I want to avoid ambiguity or problems like trying to pour from the same bucket more than once. If I'm telling another person what to do, for example, I can point to a bucket or tap on a bucket to indicate which one to pour next, or which one to pour into. So for example, I can say empty this bucket, the white arrow, into that bucket, the red arrow. And then I can repeat that and say, okay, now empty this bucket into that bucket. Empty this bucket into that bucket. Now, even with pointing, I still have to somehow keep track of the three buckets in my head in order to know which bucket to pour next. Now, to do that, I might internally use a concept like left,

middle, and right. And so now we get to the idea of a variable. A variable is a name attached to a location in the computer's memory that holds information. It is very much like having a bucket with a name on it. We can refer to the value of the variable or the contents of the bucket by using the name. So for example, if we use the names left, middle, and right to refer to our three buckets and total as the name of the fourth bucket, then we can describe the process of combining the buckets as a three-step process. So first, pour the contents of the left bucket into the total bucket. Second, pour the contents of the middle bucket into the total bucket, and finally, pour the contents of the right bucket into the total bucket. That's an example of a three-step algorithm. And by using the names attached to each bucket, we can refer unambiguously to each element of the task. Now, when we are writing programs, the buckets represent locations in the memory of a computer. In a computer program, a variable is a name attached to a location in the computer's memory, and that location holds information. It is very much like a bucket with a name on it, and we can refer to the value of the variable by using the name of the variable. Now variables hold data or information. For example, they might hold values that represent numbers. And so when we visualize the internal representation of variables in a computer, we can visualize it like this, with the names and the associated values as two columns in a table. Now names in programming languages are strings of characters. Every programming language has rules for what is a valid name. In most programming languages, a name must start with an alphabetical character, lowercase a to z, or uppercase A to Z, or possibly a character like an underscore. The remaining letters of the name can generally be alphabetical characters or numbers, or an underscore. So these are valid examples of names in most programming languages. Left, right123, middle_bucket, Total with a capital T, BUCKET in all caps, and _total. These on the other hand, are some examples of invalid names. In this case, left bucket contains a space and a programming language will actually interpret that as two separate names. 42right starts with numbers which is generally not allowed. Middle-bucket contains an invalid character because - is generally used to represent subtraction. And something like #Total also starts with an invalid character. So now let's move on to assignment. And how do we go about the process of creating a variable using assignment? So assignment is an operator for copying a piece of information into a memory location attached to a variable. Assignment is how we move information around computers. And assignment is how we can store information in computer's active memory. So when we are describing programming language concepts like assignment without referring to a specific language, so in something called pseudocode,

we will always use an arrow to represent assignment. The arrow shows the direction of information flow from the the value on the right side into the variable defined on the left. So this assignment left <- 10 copies the value ten into the memory location associated with the name left. The next statement assigns the value 20 to the variable with the name middle. And the third assignment assigns the value 15 to the variable with the name right. As we make assignments using a programming language, the computer keeps track of the variables, their names, and their values. If we want to represent the current state of the computer's memory, we can use a table similar to a spreadsheet, and we call this a symbol table. We can visualize a symbol table as having a column for names and a column for values. After the three assignments on the left, the symbol table should look like this with one entry for each variable that specifies its name and its current value. Now to complete our algorithm for combining the values in the variables together, we can use the variable names left, middle, and right, to refer to the values in the symbol table. And then specify that the variable total should be assigned the sum of the values in left, middle, and right. And given the values currently in the symbol table, total should be assigned the value 45. So to summarize, a variable is a name associated with a location in memory that can hold data or information. A name is a string of characters that follow a set of rules to find by a programming language. And finally, assignment is the operator that associates a name with a value by copying information into the memory location associated with the name. So now you can continue on to the next videos, which talk a little more about how we can use assignment and symbol tables to start to write programs.

Basic Control Flow

We're going to pick up with variables and assignments. And assignment is when we take a value and we associate a name to it. And this is a very important concept in computer science because it allows us to then manipulate those values by working with the names instead of the values themselves. So let's start with a simple example of basic assignment. So we have a name harry. And what we're saying is we're going to create an association between harry and the string "under the cabinet". And we know that harry is a name and "under the cabinet" is a string because the string is differentiated by having quotes on either side of it. Now, the way to think about what's happening here or a way to think about it is that we have a table and the interpreter is maintaining information about names and their values in a table. So in this case, we take harry and we put it in the name column. And

we take the string "under the cabinet" and we put it in the value column. And we can repeat with hermione and with the string "muggle parents". And because hermione doesn't exist in the table, we add her name to the table, and then we put the value under the value line. So now we have two names and two values in memory. So then we add the name ron and we assign the value "big family". So now we have three names in the table. Now, if we decide that we want to add ginny as another name in our memory, we want ginny to have the same value as ron, so we can write the assignment that ginny gets assigned the value of ron. And the question is, what is the value of ron? And the way we figure this out is we go look at our table and ron exists as a name in the table and it has the value "big family". And so when we're trying to decide what is the value of the right side of the assignment. We look up the name ron and we replace that name with the value, which means the string "big family". And then the string "big family" gets copied into a new entry in the symbol table called ginny. So after these four lines of code, we see four names, harry, hermione, ron and ginny. And we see four values. And notice there's not really an association between ginny and ron the names. Instead, we simply assign a ginny the same value that ron had at that line of code. Now one important concept in reading code is that order matters. So for example, consider if I took the assignment to ron and I put it after the assignment to ginny. In this case, when I get to the line, ginny gets assigned the value ron, the computer is going to look in the symbol table for the name ron. But there is no name ron in the table at that line of code. And so the interpreter will return undefined name error in that case, and my code will stop executing. So it's important that we put the definition of ron before we use it on the right side of an assignment. And then the computer will know what the value of on is when we try and assign its value to the variable ginny. Forward in the story to when our four students are learning their first spell wingardiam leviosa. And I seem to remember that harry took a while to learn his spell. So let's say he took ten tries before he learned the spell. Now the question is, we're using the same name on the left side that already exist in our table. And one way, and the question is, are we creating a new line in our table? Well, that would actually make it very difficult for the interpreter to know what the value of harry was. Because if there are two instances of harry in the table, then when I use harry in a later expression, the computer won't necessarily know which value to use. And so that creates ambiguity and ambiguity is bad when we're writing code. So instead, we want to update the existing value of harry. And so we're changing the association. And when we make the assignment of 10 to the name harry, we update the value in the table. And now I

have a situation where I'm using the name harry on both sides of my assignment because maybe I remembered. Oh, harry actually had to take one more try before he learned the spell. So the way to think about this as we always figure out the value of the right side before we update the value of the name on the left side. And when we're looking at the value on the right side, we'd look at the current value in our table. So we go to our table and we identify the name harry, and then we replace it with the value, the current value of harry. And then that lets gives us an expression. and now we want to assign the value 11 to the value to the name harry. And so we update our table. And so after executing these two lines of code, harry now has the updated value of 11. And by following that process, it's always very clear what that updated value is going to be. Now, hermione obviously learns the spell very quickly. And so we update her location in the symbol table. ron learns a little faster than harry. And so again, we look at the current value of harry, which is 11, subtract off 2. And then we use that updated value of nine to replace the existing value of ron. So now finally, we have ginny who takes just one more try than hermione. And so we replace hermione with the value 1, add those together, and then update the value of tuning. So after all of this code, we have new values in our symbol table. And if we want to then compute the total number of tries for all of our students, we can now use the values in our symbol table to discover what that's going to be. So we replace harry with 11. hermione with one, ron with nine, and ginny with two. And then the computer computes that sum, we get 23. Now total is a new name. So it gets added to our symbol table with the value 23. And what this lets us do is if we then want to print that total out to the user, we know that it's going to print 23 because we've done this evaluating of all the values of the variables. We've walked through the code linearly. And at the end we can predict what the program is going to do. So the main concepts for this video are that we have to remember that code executes in a linear order. That assignment associates a value with a name. That when we assign a value to a new name, this creates a new association. But when we assign a value to an existing name, it updates the association and replaces the old value. That the right side of an assignment is always evaluated first using the existing information in a table. And that the value of the right side is copied into the location associated with the name on the left side. So if we keep these rules in mind, we can always predict what a series of assignments is going to do and how the computer will view memory when the code is finished executing.

Assignments - Examples in Python

In this episode, we will introduce PythonTutor and Python interpreter. PythonTutor is an online Python interpreter. To use PythonTutor open the web browser and enter a URL, pythontutor.com and press enter. You will get this introduction page for PythonTutor. Let's click the link "Start visualizing your code now". We now get into the PythonTutor Interpreter. Let's choose that live programming mode, which will help us better understand naming and assignment. The window on the left is the text editor on which we can type Python code. The window is resizable and we can adjust the edge to the size we like. The space on the right of the text editor will display the table that contains the names we created and associates the values. Let's create a named harry and assign the value "under the cabinet" to it. We can see that the interpreter generates a table to show the name and associated automatically. When we create another name ron and assign the value "big family", the name and its value will be added into this table.
The left column of the table indicates a name and the right column of the table indicates values. A name in python must be assigned a value when it is created. Otherwise, we're going to get a name error. For example, we create a name ginny. But this time we do not assign a value to ginny. And you are going to notice that the interpreter that give us an error information name error at the end. And it's going to indicate that name ginny is not defined. So we have to assign a value to ginny, let's assign 1 to ginny and press Enter. And you're going to notice that there is a new entry in the table for ginny and its associated value 1. And by the way, in Python, we can we can add comments. And the comments is just a way to explain your code. For example, here we explain that a name must be assigned a value when it's created. And then we want to use that as a reminder to ourselves in the code. And to do that, we have to use the pound sign in front of that string so that when we ask you to code, the interpreter will consider the string as a comment, not part of the code. And they ..... And there ...a naming rule in python is that names in python are case-sensitive. For example, if we create another named Ginny, and this time we are using capital G here and we assign value 10 In this case, this Ginny will be considered as a different name for a different, a different name. And you can notice that there's a new entry for the Ginny with capital G in the table and the value associated with the new Ginny is going to be 10. In addition to check the table to figure out what's going to be the value associated with each name, we can actually use print function in Python to check out the value associated with the name. To use a print function, we can simply type print with a parentheses. And inside the parentheses, we

just pass their name that we want to check this value. So in this time, we want to check the value for the ginny with lowercase g and press Enter. You're going to notice that there will be an output window above the table on the right of your screen. And that's going to give us the return value for the print function. And the return value will be the value associated with the named ginny with lowercase g here. We can do that for the other Ginny with capital G. And then we can notice that the return value for the second print function is going to be 10, which is the value associated with the second Ginny with the uppercase G here. The valid name in Python can only contain letters, numbers, and the underscore characters. And the first character of a name cannot be a number. For example, we can, we can, we can define a variable we can create a name. You need to assign a value ron to it. We can notice that there is a new entry for ginny2 in the table. And the value associated with the ginny2 is a value that is associated with ron. We've basically assigned ron's value to ginny2 So ron and ginny2 have the same, have the same value. But we cannot say 3ginny. And if you do that, you're going to notice that the interpreter give you an error message and that you know that this is an invalid syntax because the first character of a name cannot be a number. So to fix it, we can easily add another character in front of their first character here so we can use the underscore. And because underscore is a valid character in, used in names in Python. So in that case, you can notice that the table create a new entry for _3ginny And value for _3ginny gonna be 10 in this case One last naming rule for Python is that we cannot assign values to keyword. For example, True is a keyword in Python which indicate a Boolean value. And if we assigned a value that says 3 and you are going to notice that the Python, the Python interpreter will complain that it cannot assign a value to a keyword and then there will be no entry for this assignment in the table as well. In addition to True False is another keyword and we cannot assign a value to False neither. And you perhaps want to know what, what are the keywords in Python that we cannot use as names? And that is a pretty good question. A handy way to identify whether a name is a keyword or not is to use a good Python interpreter, like the PythonTutor here. So we can notice that all those keyword are highlighted and so, so True, True And False here are highlighted, Those highlighted words are mostly going to be the keyword that are reserved by the Python already and we shouldn't assign value to those keywords. And we cannot use those keywords as names.

Assignments - Examples in R

Welcome to the video on R. We're looking at assignment and naming in this video. So hopefully you've had a quick introduction to RStudio, which is the integrated development environment I'm going to be using. When you open up RStudio to begin with, it will show you these three windows. I'm going to also create a new R script file. And that will give us four windows. And just a quick review. In the upper left corner, we're going to have our code file. So that's where we can write and save R code. In the lower left we have the console window. That's where we can get our, our R code will execute and any output generated by an R script will show up there. We can also do things like write simple R code and execute it immediately, which can be useful for testing out different statements and different instructions. In the upper right, we have the global environment, and that's actually our symbol table. It's going to show us all of the names and variables that we've defined and what their value is. And then in the lower right we have a window, which is the Help window. It's also where plots will show up if you're going to plot things in R. It's also where you can manage R vpackages. Okay, so beginning with assignment and naming, I'm going to be typing code into the code file. And then as we run it, we'll see things show up in the symbol table or the global environment on the right. So the first thing I'm going to do is I'm going to assign to the variable frodo the value 1. Now R lets us do assignment in two different ways. I can assign using an equal sign. I can also assign using a two character symbol that looks like an arrow. And I'm going to use the arrow for this example simply because it really does a nice job of representing the way information flows in an assignment. Information always flows from the right side of assignment to the left. And whatever the value of the right side gets copied into the memory location attached to the name on the left side. And in R a pound sign is a way to create a comment and a comment is ignored by the R interpreter. And so I'm going to say this is an equivalent assignment, using the equals sign. Okay? Now I'm going to go ahead and save my file. And I'm going to call it assign.R And you need to save our files with a capital R as the suffix. Once I've saved the file, I can execute the entire file by clicking on the Source button. And you can see that it executes the code in the console window. And then you can see in the global environment, we end up with a variable named frodo with the value 1. And so we're actually creating the symbol table as we go ahead and do the assignments. Then my next thing I'm going to do is I'm going to assign to sam the value 1. And the other way that we can execute code in R is if I put the cursor on a line of code and click Run, it will

execute just that line of code. And you can see that line of code showing up in the console. And then you can also see the result of that line of code showing up in the global environment symbol table. So now we have two variables, frodo and sam. And frodo has the value 1 and sam has the value 2. So now I'm going to assign to mary the the result of taking frodo whatever value is in frodo and adding it to the value that's inside sam. So what R is going to do now is it's going to use the global environment symbol table to look up the value of frodo, which is 1, and look up the value, of sam, which is 2 And it's going to compute their sum, which is the value 3. And then it will take that value 3 and assign it to mary. So if I click Run on that line of code, you can see it adds mary to the symbol table because we've assigned a value to it, and it has the value 3. Now the assignment does not create a connection between frodo, sam, and mary in this case. And we can see that because if I assign frodo a new value of 10 and run that line of code. It changes the value of frodo, but does not change the value of mary. And that's because assignment copies the value from the right side into the, into the memory location attached with the name on the left side. And so on line 6, when we computed the value of mary, the value three was computed from the right side and copied to. the location associated with mary And so then when we later on change the value of frodo, it overwrites the old value of frodo because we can't have two symbols with the same name in a global environment symbol table. So it overwrites the old value of frodo and gives frodo the value 10. Okay? Now, what happens if I try to assign to a variable, another variable that doesn't yet exist. So in this case, I'm trying to assign to mary which does exist, the value that's in pippen. But unfortunately, pippen does not exist. And so the R interpreter tells us this down in the console window and it says error object pippen not found. And that's because pippen doesn't yet exist in the global environment symbol table. So if I first assign to pippen a value like 5, and then I run the reassignment of pippen to mary. Then now both of those execute safely and after assigning 5 to pippen, it shows up in the global symbol table. And then after assigning the value of pippin to mary, mary also takes on the value five. If I then modify the value of pippen to be 6, it does not change the value of mary because the assignment of pippen to mary did not create a connection between those two variables. It's simply took whatever value pippen was at the time, and copied that value over to mary. When we reassign the value 6 to pippen it does not affect the value for mary. Now, we can also do expressions that look a little funny in mathematics where we say pippin gets the value pippen + 1 But in computer science, this is perfectly fine because we evaluate the value of the right side first

and the value of pippen is 6 to start And so we take that value 6 and we add 1 to it to get 7. And so if I run this code, pippin will take on the new value 7. And so then we can finish. I'm going to go ahead and clear the global environment. We can do that by clicking on the broomstick. And that's like we're starting fresh. None of the code has been executed. And now I'm going to go ahead and step through the code line by line, the way the R interpreter would do so. And so we first create the variable frodo with the value 1. Then we create the variable sam with the value 2. And then we look up the values of frodo and sam, which is 1 and 2, add them together to get 3, and that value gets copied into the variable mary. Then we reassign the value 10 to frodo. Then we assign a new variable, pippen and give it the value 5. Then we copy the value of pippen to the variable mary, and then we reassign the value of pippen to 6. Finally, we take the current value of pippin, which has 6, add 1 to it, and reassign that back to the variable pippen And after rerunning all of this code, we've rebuilt the global environment symbol table, and it tells us what all of the values are for these variables. If I were to do anything with them on line 15. So for example, if I were to print the value of frodo, it should print 10 to the console. And in fact it does down here. So by understanding the symbol table and understanding how to read it, I can always predict what an assignment is going to do or what a print statement is going to do when I print out one of these variables. So to review, we learned how to create variables by assigning a value to a name. We can see those names and their values in the global environment symbol table. R will always execute code in the linear order, starting at the top of the file and moving down. An assignment always makes a copy of the right side and puts that copy into the memory location attached to the name on the left side. And so that's an introduction to assignment and naming by using R and the IDE RStudio.

Basic Types & Operations - Python

In this video, we'll explore basic types and operations. Each programming language makes it possible for you to work with several different types of data. Typically, some of these types are very elementary, while others are more complicated. In Python some of the elementary types are integer, float, which is for decimal numbers, and strings, which are for character string data. Some of the most commonly use complex types are lists and dictionaries, which we will not look at in this video. It's important to think about what type of information you will be working with, because that helps you determine

what kind of operations you can carry out and what sort of results you should expect. For example, consider the following in Python. We can do simple addition of two integers. We can add a floating point number and an integer. We can add an integer and a floating point number. And we can add two floating point numbers. And we see that if we add the two integers, we got an integer result. If we add two decimal numbers, the result will be decimal. If we add one integer and one decimal, we still get a decimal result. This is because Python does not want you to lose any information that is in the decimal portion of the number. For example, if I do this addition, I really want the answer to be 3.5. It would not be helpful if the answer Python gave me for that was 3. We will get the same behavior if the values are stored in names instead. For example, if I have 4 dogs and I have 5 cats, then I will have 9 pets. When this addition is carried out, the value of dogs is retrieved from the symbol table, the value of cats is retrieved from the symbol table, and then the addition operation is carried out and the result is stored in the variable pets. If I have 6 cookies, but somebody ate half of one of my candy bars, then the total number of treats I have is 7.5, seven and a half. The floating point addition operation is carried out because I have an integer stored in cookies and a decimal number stored in candy bars. It's important in each programming language to know how different operations will behave on different types of data. We have seen here what the addition operation does with integers, with decimal numbers and with combinations of those two. The addition operator, the plus symbol, is also defined for string data, in which case it carries out a string concatenation operation. If I have this string, 'cats ' with a space at the end, and I add it, so to speak, to the string 'dogs', I get cats space dogs This will also work if those values are stored in names. So I'll make animal1 be 'cats ' with a space and animal2 be dogs. And now if I use the addition symbol, the addition operator, we get that string concatenation. I can get a little fancier and in this case I'm going to create a new variable, myPets and what I'm going to do is concatenate the value stored in animal1 with the string 'and ' with a space at the end and concatenate that with the string stored in animal2. And so we can see that myPets is now the string 'cats and dogs' What would happen if I tried to do something that is not defined in Python? Like I tried to add 'cats ' + 6. I get a nice big error message telling me that I can only concatenate a string with a string. That I cannot concatenate string with an int or an integer. So Python could tell that I was trying to apply this plus symbol to a string value and an integer value. And it says, forget it. You can use that symbol, that operation. with two numbers, you can use it with two strings, but you are not allowed to

use it with one string and one number. The multiplication operator is also defined for strings in Python, of course, we know what behavior to expect if we apply that to two numbers. We see it with integers. We can see it with a float and an integer. and we can see that with two floating point numbers. It also works with values that are stored in variables. So if I have 5 reams of paper, and I have 500 sheets in each ream. In the same way that I did with addition, I can use those names, those variables, in computation. and compute the total amount of paper I have, which is reamsOfPaper times sheetsPerReam And as we would expect I have 2500 total sheets of paper. If I use that same multiplication operator symbol with a string and a number, it carries out what we call a string multiply operation, creating a new string that has multiple copies of the original. So if I take the string 'fun' and I multiply it times 5, I get a lot of fun. And as we've done in our other examples, if I assign a value to greeting, I can do multiplication of that with a number. And the order does not matter because multiplication is commutative. And I could take that result and assign it to a new variable. And I can see the 7 copies of the string 'Hello ' Of course, it probably will make sense to you that the other arithmetic operators are not defined for strings so there's no way to use subtraction or division with string values in Python.

Introduction to Functions

This movie is to introduce the concept of functions.
We organize code by naming groups of instructions.
I'm going to draw these two purple rectangles
with a fairly lengthy set of instructions.
First of all, my pen on the paper and I
move forward by 100 steps, and I turn left in preparation
for my second edge, I move forward by 200 steps,
turn left, 100 steps, turn left. 200 steps turn left.
I've finished my first rectangle.
Time to pick up the pen, move to new location and
put the pen down again. Then I can repeat the process.
I'm going to move forward by a 100 steps and turn left 100,
one hundred, two hundred. And I've completed my goal.
I've drawn two identical rectangles with
two identical sets of instructions.
This is a problem. Whenever we have
a repeated blocks of identical instructions,
we have two problems.
One is that it makes our code longer and

long code can be hard to read and
it's harder to absorb the overall point.
Also, we have an organization that is not explicitly
demonstrated in the code. Our solution is to group
the instructions together explicitly
and give them a single name. In this case, I'm going to
take one of these groups of instructions and call
it draw_rectangle, and then I can use it twice.
I'm going to run those instructions using that name,
rather than simply having two copies of the same code.
A function is a named block of instructions with a
particular purpose. Here I demonstrate
how we might draw, make our function that
we'll call draw_rectangle. It has a copy of all our
instructions in there, and we write it just once.
Then in our main code, we refer to it simply by name.
We say draw a rectangle, then we can pick up a pen,
moved to a new location, put down the pen,
and draw a rectangle again. So our organized instructions
now include a function with a name,
the definition where it has all of those
instructions in it. And then we have
overall code that is easier to read. Thank you.

Calling Built-in Functions

This video is about calling built-in functions. To take advantage of functions, they're part of programming languages, we need to call them. print is a function that is common to almost all programming languages. And here is an example of a line of code that calls the function named print, giving it the input "Hello, World". I just used the term call, but there are three words that are equivalent. We can, we can say the function is called, that it is executed, or that it is invoked. And I like to think of this as a two-step process. First, the value of the input is sent to the function and then the function executes its instructions. This code results in the message "Hello, world" being printed to the screen. Some functions return values. Many programming languages come with libraries that perform mathematical functions and many of those return values. Let's consider the function here that will compute the average of all the numbers in a list. This example, I've got the list 5 6 7. We're going to compute the average. I'm expecting the output to be 6. Let's step through the process of executing this function. So I have my line of code with the input. When we call the function. Now

there's three steps. The input is sent to the function. The function executes its instructions, and then the value is returned from the function. And we illustrate it like this. So step one, I'm putting that value in, sending it. Step 2, the function is executing. I'm not actually showing it here on the screen because I don't know what the code is in that, I don't need to. All I know is it's doing its thing. And then a value is returned. And I'd like to illustrate that by showing okay, the function is executed, which means it's no longer running. We cross it out and we'll replace it by the value that that function returns. And it's really useful to have functions that return values, but we need to do something with that returned value. So I'm going to repeat the example now but I'm going to put it in the context of code that is a little more useful. So this code here calls compute_average, but notice it's on the right hand side of an assignment statement. That arrow means I'm going to assign the return value from compute_average to a variable named mean. So now this is imbedded in a two-step process. First I'm going to call my function compute_average. And second, I'm going to take that return value and assign it into a variable. So I've got my symbol table all set up on the right. I'm calling my function, sending it input. Now it's doing its thing computing 5, 6, 7, getting, it's gonna get me 6. It's done. It says, I've finished, my value is 6. So now it's time to finish the assignment. Now I have an assignment statement in which 6 is going to be assigned to the variable named mean. It gets added to the symbol table. And voila, we can use a variable named mean now to print out or do whatever we want to do with the average of the list 5 6 7. But there's one more thing that often happens and that is that when we call function, we're not just writing the list value directly in there. We might have that list, or maybe a data frame if this is R, in a variable. So I want to go through this example one more time showing how we take the input and pass it into the function. So again, I have pretty much the same example. I'm going to compute the average of the list 5 6 7 and put it into a variable named mean. This time though, my symbol table already has a variable named nums whose value is a list 5 6 7. And, as before, first I need to call the function and then I need to finish the assignment statement. So here step 1 means instead of just trying to pass in some variable I go to the table, I look up the value, I take that value and that value is what gets sent to the function. Then compute average does its computation. And when it's finished, it says, I'm done. Here's your value, it is 6, and then we finish the assignment statement. Thank you.

Calling Functions in Python

This video is about calling functions in Python. I'm going to use the print function as my running example. Most programming languages have something akin to print. Most functions accept some sort of input and the technical term for the input to a function is an argument. Here I show print with the argument "hello". An argument can be a literal value such as number 4 or the string "hello" or it can be the value of a variable. So here I have print and I'm calling it with the variable name num1 in between the parentheses. Or an argument can be something more complicated, such as the result of an expression. Over here, I'm calling print and the argument is the result of adding the value of num1 to the value of num2. Let's step through that code in PythonTutor. Okay, I have first an assignment statement in which I assign a literal value 42 to variable named num1. And then another assignment statement where I assign the literal value 31 to a variable name num2. And then I call print, passing it the argument "hello". And hello appears. Next I call print, and I'm passing it the value of the variable num 1. So what Python is doing is looking at the value of num1 and passing in 42. So it prints 42. And finally, I have a mathematical expression the results of which becomes the argument to my function. Python looks up num1, gets 42, looks at num2, gets 31, add them together, gets 73, and then prints that. Many functions have more than one argument possible. And print is one of those. You can print more than one thing by passing in more than one argument. And in the examples that I have here, I'm using what's called positional notation. So Python prints the value of the first argument first then the value of the second argument, second, and the value of the third argument third. In this example print I print "Nums are" then the value of num1 and the value of num2. In this example, I print "Their sum is" and then the the value of num1 added to the value of num 2. Let's step through that code. We have the same assignment statements we had in the previous example. Then we print "hello" with just one argument. And then we print "Nums are" with two additional arguments. And each one's evaluated separately. We look at the literal value "Nums are" then we lookup the value num1, then num2, and it prints all three. And finally we print "Their sum is" and just like in the previous example The value num1 is looked up , the value num2 is looked up, and then the values are summed and so we print "Their sum is 73" In many functions the parameters have names, and we can call functions with what are called keyword arguments. And that's where we refer directly to the name of the function, sorry, the name of the parameter. So here's the documentation for print and we can see that takes in a bunch of things that we needed printed. But then it has a

special a parameter named sep for separator. And by default, its value is just a space. And the documentation here tells us that it will separate each of the objects that's being printed with whatever is the value passed in here. To specify that I will use the keyword sep then. So here I have, I am calling print with num1, num2, and then I'm indicating that as a separator, I want to use the plus sign. This is called a keyword argument because I put the name of the parameter first and then an equal sign. And this is a fairly complex example because I'm using two positional arguments and then a keyword argument. Down here I'm attempting to use a keyword argument and then a positional argument, but this is not legal. Any keyword argument needs to be followed by all keyword arguments. So you can put positional arguments and keywords, but not the other way round. Let's look at PythonTutor. Here we have the same assignment statements. Oops, but I grab the code that doesn't work. And now we can see the error that is reported to us. It says positional argument follows a keyword argument. Can't have a positional argument after a keyword argument, it simply won't work. So we comment it out. And then we run. And we can run our assignment statement and our next assignment statement. And then we print, it looks up the value of one. It looks up the value of num2, and it says I've got a separator for you. So it's going to write 42 plus 31. And we are done with that. Let's move to a more complex example. We can call a python function with the output from another function. So we talked about how we can have more complex things such as mathematical expressions as, as arguments. But we can also actually take the output of another function and pass it in. So here I'm using three functions that are built into Python. len which provides the length of a list, print, and sum which sums up the values of the elements of a list. I first make a variable named nums that has the value [5,6,7]. I call length len function with nums as its input and its output is the value is the length. I'm going to store that in a variable named N And then I can do a print statement just like we did in the previous slides, where we print the length with a literal string that says what it is we're printing and then we have a comma and then we have a variable name. You look that up, and that's when the value that's printed. But we could do it directly we could skip the whole process of using the variable and instead just take this function call and put it here. Anyway, Python will execute, It will call print and then evaluates what arguments are. Because that's a literal I know how to do that I'm going to pass that right along in with this I got a call function. I pass in the list nums which I had to look up And sorry, I have to pass in a list that nums refers to, which is 5, 6, 7. The value 3 is returned.k. Finally, I can, I include the output of functions in expressions. So in this

case, sum and len both return numbers So I can divide one by the other and that gets the mean, and I can print that out. Let's look at PythonTutor. Have my assignment statement, nums is the list 5, 6, 7. And then I assign N to have the number of elements in nums. Then finally I print then accessing a variable, that is three. And I print again where I'm evaluating len directly in there, I get the same thing. And then I call sum passing in nums. It gets me a number. I call len pass in nums That gets me another number. And I do division. And then I print out the result, which is 6. Thank you.

Calling Functions in R

This movie is about how to call functions in R. Let's begin with a simple helpful function, print. To call a function, I need to supply it with one input and that is what needs to be printed. The technical term for this value right here is called an argument. That's what we pass into a function. And here I'm using a string literal "hello" and it prints out "hello". Now let's switch to a function that has more possible inputs, more arguments. Another technical term: when we look at the help, the, which often includes the definition of the function, we call it a parameter. And when we, when we call the function, execute the function with the value we pass, it's the argument. And whenever we call a function, we need to have the arguments match up with the parameters. So for sequence where we're going to focus on these three parameters:from, to, and by. So the starting point, the stopping point, and the step size of our sequence. We need some way of having R understand which argument is for from, which is for to, and which is for by If I supply it with seq(1, 5) then it's going to match up this argument 1 with the parameter from, because it's going to go by position, the first argument is going to be associated with the first parameter and the second argument, 5, is going to be associated with the second parameter, just to, t-o. So that should produce the sequence 1 2 3 4 5 which it does. Now this sequence this function here, seq has what are called default values associated with each parameter. So if you don't supply a value for from, it's going to use 1. If you don't supply a value for to, it's going to use 1. If you don't supply a value for by, it's going to calculate the step for you. And these integer sequences, it's going to be 1. So the implication here is that I can call seq, without any input and it'll just give me 1. And notice even though it's got five parameters because they've all got default values, I can supply a subset of them. So for example, if I wanted to do 1 3 5, I could do the 1 and the 5 that I've already done and make my step size 2. There's another way that R can match up arguments with parameters, and that is by name.

You can actually include the name when you're calling the function. So I could say this from is 1, to is 5, and by is 2. And I get the same sequence. And the beauty of this method of calling is that order no longer matters. So I could say, well, it's more natural for me to think about it as start, step, stop. I'm going to write it like that. This also has the advantage of being very readable. R can handle a mixture of arguments that need to be matched by position with arguments that are matched by name. So I can simply say 1. And then I could say by is 2 and to is 5. We have the same thing. R is rather sophisticated in its in how it does its matching up. I don't really recommend doing things out of order like this, but, but we could say from is 1 and then give it the number 2. And then say to is 5. And what R will do is it will say, I'm matching from by name and to by name. This value right here, the number 2 doesn't have a name associated with it, so I need to do it by position. So what I'm going to do is scan through all my parameters and say, have I already match this by name? No, have I already match this by name. Let me start here. I already mentioned from my name, yes. He managed to buy it. No, I haven't. So this number 2 Sorry. I have already mentioned to you by name. It's by that hasn't been matched yet. So this number 2 must be associated with by. So this must also be start at 1, go to 5 and step by 2. And it is. I recommend using the match by name. It just reduces confusion. So we've, all the examples that we've had so far are having arguments that are either numbers or strings. So these are what are called literal values, I should write the value in and pass it to the function. But you could also use variables. So let's set up some variables. I have my starting point, which is 1, by stopping, which is 5, and my stepping, which is 2. And I could call something like this, from equals starting, to = stopping. And by equals stepping Further, we could use something more complicated, not just refer to a variable. But we could actually use an expression involving variables. So let's use a simple mathematical expression. Let's say we want to go to bigger. Forget stopping at the stopping point. I'm going to go one big step further and add 2 to that. So now my argument is actually the result of the expression stopping + 2. So that should be 1, 3, 5, 7. And in fact it is. So we've gone over so far how to match up arguments or parameters, that's either by position or by name. And what kinds of values or arguments can be. They can be literal values. They can be the values of variables, they can be the values of expressions. And I have one more example I want to show. And that is the fourth kind of thing an argument can be is it could be the output of another function. So one simple example is print needs an input. I can simply give print the output of calling seq. So here the argument to print is the output of my seq function.

Let's make a example. we had in the earlier movie where our sequence was 5, 6,7 From is 5, to is 7. So nums is 5 6, 7. Awesome. Let's just show you that we can type mean nums. That's a built-in function at num pi that R has or instead of saving the sequence in a variable named nums, I could just build it on the fly and pass it in. And there it is. The mean here took this output of calling this sequence function freshly rather than just from a variable. Thank you.

Defining your own functions - Python

In this video, we talk about defining your own functions. Before you write your own functions, it is helpful to understand that when a function is called by the main program, control is passed from the calling program to the function, the function code executes, and when that process is complete, control is returned to the calling program. We can think of this as program execution taking a bit of a journey from the calling program to the called function and back again. While the function that was called is executing, the program that called it does nothing. Its execution is paused until the function completes and then when the function returns control, the calling program can continue executing at the point just beyond the function call. Control is always passed but there's also the issue of whether or not information is passed between the calling program and the function. When we consider this aspect of function usage, there are four possibilities for the connection between a function and the program that calls it. In this video, we work through all four of those. This should give you an understanding of the different ways in which you can write functions to do some of the work required in programs that you develop. We'll start with the simplest situation, a function that has no parameters, so it receives no information when it is called and it returns no information. A function like this would be useful if you wanted to create a nice user interface for your program, perhaps by displaying a menu of options for the user. For example, let's consider a function that displays the menu for an inventory maintenance system. We can call that function with a very simple statement shown here in the main program. Just like when you call a built-in function, execution of the call to the menu function causes control to pass to that function. The function will execute one line at a time and when the return statement is reached, control goes back to main. main starts up after the call to menu, which in the example is the print statement that prints OK. We can see this process in Python Tutor. This is the same code that was on the slide, and we'll

visualize the execution. The first thing that happens is Python indicates that it knows that the menu function exists and it knows that the main function exists. And then it calls main. main in turn calls menu. And now each statement of menu will execute. We can see here on the right that the menu, is going to be printed out one line at a time. And that keeps going until execution hits the return statement. return basically says send control back to whoever called the function. In this case, control will go back to main. You can see that menu is getting ready to return and it says that there's no return value, that the return value is None. And so when we execute the return, we can see that control is now back in main at the statement immediately after the call to menu, which is the print statement, and that print statement executes, giving OK in the output. The next kind of function is one that receives some information but does not return any information when it finishes executing. This setup is common. If we write functions that are used to generate nicely formatted output. For example, if we stick with our inventory example, we might have a function like this prettyPrint function. When prettyPrint is called, three pieces of information, present in the function call are passed to the three parameters in the function definition, giving those parameters values that are used during function execution. We can see this unfold in Python Tutor. So we see that prettyPrint is defined, main is defined and then main calls, main is called main calls prettyPrint. And we can see that the assignment of values to parameters was done at the time of call. So this string "no name laptop" was assigned to the parameter item in prettyPrint. 25 was assigned to quantity, and 750 was assigned as the value of price. And control has been given to prettyPrint. So now prettyPrint will execute one line at a time. The first line of output says that in inventory we have 25 of "no-name laptop" because 25 was the value passed to quantity and "no name laptop" was the value passed to item. In the second line price will have the value 750, and the total will be computed from this multiplication between quantity and price. So information came into prettyPrint when it was called, it does its output operation, no information will be returned. And so again, we see that the return value will be None and control goes back to main, and because there is no other statement, that's the end of the program. Next, we'll define a function that has no parameters, so it receives no information, but it does return information. You would use a function like this to perhaps display a menu as we did earlier, and also get the user's choice and return it to the calling program. Calling this function is a bit different than when we called our original menu function because now we have to have a variable that can receive the value returned by the function. The way to think of this

is that the call to userOption on the right-hand side of this assignment will be replaced by the value that is returned from the function. And then that value will be assigned to the variable on the left, in this case the variable action. And then I will verify that that worked by printing out the value of action afterward. userOption is defined and we see it expects no values because it has no parameters. main is defined and then main is called. So here we are at line 14, the assignment statement. The assignment cannot take place until there is a value on the right-hand side. And there will be no value on the right-hand side until the function userOption executes. So the call is made to userOption. The menu is printed out. And here we are at line 9 which is the place where the user is asked for input. And we'll enter our choice. If you were running in regular Python, not in Python Tutor, you would just type your choice and hit Enter. Here I'll type my choice and click Submit. And now we can see that we're at line 10, which is the return statement, which will return control. But we're also returning the value of choice. That is, we are returning the value associated with the variable choice in userOption, which is that character 'a' that I entered as my, my option. This means that the return value, instead of being None as it was earlier, is now the string 'a'. Control will go back to main and we can see now that the assignment has taken place and action, the variable action, has the value 'a' Next the blank line will print, and then the message confirming the user's choice, the user's desired action. Finally, our last category of function is those that both receive information and return information so they have parameters and return values. These are functions that we generally use to do calculations. The vast majority of functions you write will be this kind of function. Information is passed to the function. The function carries out some calculation that is relevant to the problem being solved, and then it returns the result. We can see this with a variation on our earlier prettyPrint function. This time, I'll leave that printing in main but compute the value of that item in a separate function. We'll head to PythonTutor once again to visualize execution. The value function is defined. main is defined. We call main. main will assign 25 to quantity, "no name laptop" to item and 750 to price. And then it calls a print statement, passes it some information, so the print statement generates a statement about the the quantity of "no name laptop" we have. And then the program is ready for the next print statement. But the second print statement cannot actually execute until it has all of its component parts. It won't have the last piece of information until it calls the value function and value returns a result. So it calls value, passing it quantity and price. And we can see that the value function now has

its own memory space and its own variables for quantity and price. Generally, you do want to use the same names in a function as you used in the calling program because it makes your code more readable and understandable because the quantity variable in value is fundamentally referring to the same thing as the quantity variable in main. Our value function is so simple that it simply does the computation needed in the return statement and returns the resulting value. So we can see that the return value will be 18750. And when that value is sent back, it will be plugged right into this spot in the print statement, and the print statement can execute. Finally, we'll look at an example where we have two user-defined functions and one calls the other. Basically, we have both our earlier prettyPrint function and our value function. Originally prettyPrint did a computation directly in the second print statement. But in this version I have modified prettyPrint so that it calls value to do the computation. So the main program sets up the call to prettyPrint, passing it three pieces of information. One for the item, one for the quantity, one for the price. So I can see that prettyPrint now has its three pieces of information. And because it has those three pieces of information, it has the information it needs in order to call value function. So the first print happens and then the second print statement in prettyPrint will call the value function. It is passed, value is passed the quantity and price information. And we can see once again, it can return the 18750 which will come back to prettyPrint and get printed out in the second print statement. That wraps up our tour of the four different kinds of function setups you can have when you write your own code, when you write your own functions. So when writing your own functions, you always should consider the following. What part of the program's job can you compartmentalize into a function? You want to think in particular about operations that need to be done more than one time, so writing that code once, but using it from different parts of the program is helpful for the development of your program. You want to think about whether the function needs to receive any data in order to do its job. And, if so, exactly what data does it need. And you should think about whether the calling program needs to receive results from the function which is connected to the job that the function is doing. So you need to be clear on how the results will be used by the calling program. You should answer these questions and think about the flow of control and the flow of data before you actually write any functions. Thinking about these questions, answering these questions, while you're developing the design of a program will save you a lot of trouble later on as you're writing code and trying to get your programs to work.

Defining your own functions - R

In this video, we talk about defining your own functions. Before you write your own functions, it's helpful to understand that when a function is called by the main program, control is passed from the calling program to the function. The function code executes, and when that process is complete, control is returned to the calling program. We can think of this as program execution taking a bit of a journey from the calling program to the function and back again. While the function that was called is executing, the program that called it does nothing. Its execution is paused until the function completes. And then when the function returns control, the calling program can continue executing at the point just beyond the function call. Control is always passed. But there's also the issue of whether or not information is passed between the calling program and the function. When we consider this aspect of function usage, there are four possibilities for the connection between a function and the program that calls it. In this video, we will work through all four of those. This should give you an understanding of the different ways in which you can write functions to do some of the work required in programs that you develop. Typically, in R programming, you're less likely to see all four of these, then you might in other languages. But this will give you a sense of what the possibilities are, which will be useful if you change to other languages in the future. We'll start with the simplest situation, a function that has no parameters, so it receives no information when it is called and it returns no information. A function like this would be useful if you wanted to create a nice user interface for your program, perhaps by displaying a menu of options for the user. For example, let's consider a function that displays a menu for an inventory maintenance system. We can call that function with a very simple statement that I have here on the left but we'll see that at the end of the code when we go into R. So let's see what happens when we run it. Here's the same program using RStudio as the development environment. I'll go ahead and run it. And we can see that all of the menu options are displayed as a result of the call to the function that was made at line 11. And then we see in the output the OK which I put in just as a confirmation that the function finished, finished execution and we reached line12. Just like when you call a built-in function, execution of the call to this menu function causes control to pass to the function. The function will execute one line at a time. When the end of the function is reached, control goes back to the calling program, which continues executing after the call to menu, printing out OK at the print statement. The next kind of function is one that receives some

information but does not return any information when it finishes executing. This setup is common if we write functions that are used to generate nicely formatted output. For example, if we stick with our inventory example, we might have a function like this, which is designed to provide nice output. When prettyPrint is called, three pieces of information present here in the function call are passed to the three parameters in the function definition, giving those parameters values during the function execution. Again, we can see that in R. Here's the prettyPrint function. It's set up to receive three pieces of information, the item name, the quantity, and the price. In the call, we pass values for those three parameters. We pass these three pieces of information. In the function, I'm using the R paste command, a call to the paste function, to do string concatenation. And when the function runs, we see the printing of the two lines of information. So again, control at line 8 is sent to the function, along with the three pieces of data, the three pieces of information. The function executes one line at a time, using the provided information as necessary, completes its job, and returns control to line 8 where the call was made. Next, we'll define a function that has no parameters, so no data comes in, it receives no information when it is called, but it will return information. You could use a function like this to display a menu, as we did earlier, then get the user's choice and return it to the calling program. Calling this function is a bit different than when we called our original menu function because now we have to have a way to capture the value returned by the function. We handle that by putting the call to the function on the right hand side of an assignment. The way to think of this is that the call to userOption will then be replaced by the value that is returned from the function. And then that value will be assigned to action on the left of the assignment. And I can verify that it worked by printing out the value of action after the function execution is complete. We'll see this in R. In my userOptions function. I am using an explicit return statement at the end of the function at line 10, although in R, it is not necessary to have an explicit return. If I took line 10 away, whatever line 9 computes is what would be returned by the function. Since many of you watching this are new programmers, however, I think it's helpful for you to have the explicit return statement. Now, I'm going to execute this and at line 13 the assignment cannot happen until the call to userOption executes and the return value becomes available on the right-hand side of that assignment statement. So I've hit Run. We can see in the console window that execution basically paused here at line 9. And the readline is waiting for the user to enter their choice. So I'm going to enter 'r', type 'r' and hit the Return key.

Once I do that, line 9, the readline, is complete. It assigns whatever I entered to the variable choice, and then that value is returned to the call at line 13. We can think of this right hand side of the assignment, the call to userOption, as being replaced with that character 'r' which was the return value. And that value is then assigned to action. And we get confirmation of that when line 14 executes and the user's choice is printed out. Our last category is functions that both receive information and return information. So they have both parameters and a returned value. These are functions that we generally use to do calculations. Information is passed to the function. The function carries out some calculation that is relevant to the problem being solved and then returns the result. We can see this with a variation on our printing operation. In this case, I've put the printing steps in a main program, but I'm computing the value of the item, in this separate value function. The function call has two parameters, so we pass two pieces of information, one for the quantity of an item and one for the price. The function will return the total value of those items. And so we can see that my main program consists of some setup where I assign values for quantity, item, and price, and then I set up my print statements. I set up the information that will be printed by creating the string concatenations. And in this second one, I have a call to value. We can see this a little more easily in R because the, the way the line wrap is not used here. I did need it on the PowerPoint slide. So we can see here I'm setting up values for quantity, item, and price. I set up the string that will be the first line of output. And when it's time to build the second line, I have the call to value, which will pass the value of quantity, the value of price. Those come to the function. The computation is carried out, the value is returned. It replaces this function call. And there's an actual number that can then be used as part of the output. And so when I run this I can see that that place where we had the function call was effectively replaced with the result of that multiplication 18750. So I want to look at one final example where I have two user-defined functions and one calls the other. The main program starts here at line 13, where I initialize quantity, item and price, and I make a call to prettyPrint. So this looks a lot like my original prettyPrint example. But when I look at this version of prettyPrint, it's not doing the computation in, in place. It's calling the value function. So I've written two functions, my prettyPrint function and the value function and my prettyPrint function, will call the value function. So the main program sets up the initial values. It makes the call to prettyPrint, passing three values which come into prettyPrint as the item, the quantity and the price. And then prettyPrint will call value, passing

the quantity and the price, value will do the computation. It returns to prettyPrint, which finishes its job, and returns control to the main program. And in this case there is no additional work done by the main program. So this gives you a sense of how you can create the same kind of sequence of function calls and operations that you have when you're using built-in functions in R as well. So that wraps up our tour of the four different kinds of functions setups you can have when you write your own functions. When you write your own functions, consider the following things. First, what part of the program's job can you compartmentalize into a function? You want to think in particular about operations that need to be done more than one time. Writing that code once, but using it from different parts of the program is helpful for the development of your program. Second, you want to think about whether the function needs to receive any data in order to do its job. If so, exactly what data does it need? And you want to think about whether the calling program needs to receive results from the function. And that's related to what the job is that the function is doing. So you want to be clear on how the results will be used by the calling program. You want to answer these questions and think about that flow of control and that flow of data before you actually write any functions. And that will save you a lot of time and a lot of trouble later on as you're writing code and trying to get the program to work.

Memory Models for Sequences

In this video, we're going to discuss sequence memory models. You should already know some simple data types. The most common are numbers in their various forms like integers and floating points or floats, and Booleans, simple true, false statements. There are some more complex data types. Some of these are composed of elements and you can think of elements as things that you can go through one at a time. And you may already know one or more of these types of data. For example, strings are a very common sequence. Things like "harry", "ginny weasley", and "he who shall not be named". But there are also things like lists, or vectors, and data frames. And we'll cover some of these lighter. We'll start with strings. Strings are a good example of a sequence. So what are the properties of a sequence? Sequences have elements, and when we're thinking about strings, those elements are the individual characters in a string. For example, 'harry' is composed of the letters h, a, r, r, y Sequences also have length. The name 'harry' is five characters long, as you can see from the way I broke out the letters just above, if we look at a string

like 'ginny weasley', that's 13 characters long. it's important to remember that in strings, the space counts as just another character. And sequences also have order. The order of the letters is not accidental. There is very much a first letter and then a second letter and a third and so on and so on. So if we consider the name 'harry', it should be no surprise that the first letter of 'harry' is the letter h. The second letter of 'harry' is a. And so on. The thing about these order positions of the sequences is that we can use them, we can use them to access individual elements or sub-sequences. And depending on the language, those ordered positions are numbered slightly differently. For example, in languages like Python, Java, the first position in any sequence is position 0, followed by 1, followed by 2. In a language like R, the first position is 1, followed by 2, followed by 3 and so on. Other than this difference in starting position, the way that we use those positions is exactly the same. So for example, if I have a sequence in R, This would be, for example, a text string. Here it is the name harry. Then I number the positions of each element of that string, starting from the number 1, that the letter h is at position 1, the letter a is at position 2. We call those positions index values or indices. And it's worth noting that for any language that starts counting at 1 for a sequence, always know the index position of the last element because it's exactly the same as the length of the sequence. It's exactly the same in Python with that one slight difference of the starting position. Here's the same string. But in Python, that letter h would be at index position 0. The a would be at 1, r will be at two. And when we're dealing with languages like Python, it's always worth remembering that the index position of the last element is the length minus 1. Harry has a length of five, the index position of the y, the last letter is 4.
And strings are not the only sequences you'll come across. Lists or vectors are also a very common sequence and they have some particular properties. For example, lists a heterogeneous, you can store all kinds of data in them, but they still ultimately obey the same rules. For example, lists have elements, lists have length, and lists have order. I've given you an example here of a Python list comprising five integers, one through five. And if I were to represent that, you can see the five integers 1, 2, 3, 4, 5. And here I'm representing them with Python indexing starting at 0. So at index position 0 is the integer 1. At index position 1 is the integer 2. And again, the index position of the last element is the length of the sequence minus 1. There are 5 elements in the sequence. It has length 5. The last index position is 4. And just to touch on how these things are represented in memory, there is a difference between them. Strings are represented directly. Here, we're showing you the symbol table

that you've seen before. On the left-hand side are the names as they are represented in the computer. And on the right hand side is the value associated with that name. When we create a string, for example, harry, we associate the value harry with the name of the variable, in this case name, in the table directly. But other types of sequences and indeed other data types are represented slightly differently. Here you can see the name of my variable, numbers, appears in the left hand side of the symbol table. But on the right-hand side, instead of representing that information directly, what I'm doing is creating a vector of the integers 1 through 5. Instead of directly placing that in the table, I have instead a reference to those values. And that's going to be important for some things that we do later on. So things like strings, we represent directly, things like lists and vectors we might represent indirectly by using a reference. And that's what that arrow is there to represent. We can access elements of a sequence using those index values. So for example, if we want to access a substring, then we need to know the index values of the substring we want to extract. In the case of getting the letter h from harry, we need to know that in R, that's the first element. So that would be index position 1. Similarly with numbers, the list or the vector, if we want to access the second element of that list, we need to know it's index number. That would be 2 in R or 1 in Python. And there's one further piece of information. We cannot change the elements of a string because strings are immutable. So what does that mean? It means that just because I know the first element of harry is the letter h, I cannot change the first element of harry to be a different letter, will either get an error or will end up replacing the whole association between the variable name and the value. Conversely, we can change the elements of lists or vectors. Lists have a property in that they are mutable, we can change them. So if I try to change the second element of this vector, that value that is currently 2, to 99, it will actually change. And in R I can do that by referencing the variable, giving it the position in square brackets that I want to change and then making an assignment. And that will actually change the second element of that vector. And that's going to conclude us talking about the memory models of sequences for now.

Memory Models for Dataframes

So the most important components of a computer are its processor and memory. The processor is the piece that does all the actual

computation. And memory is where both the programs and the data that is processed by these programs is stored. In order to execute a program, what the processor does is it gets one program instruction from memory. It then retrieves all the data from memory that are specified by this instruction. It manipulates the data according to the instruction and then writes any possible results back to memory. And then it just does this over and over again until the program is complete. So a computer essentially has a processor that's interacting with memory. This memory that I've been talking about is sometimes also called working memory because it's not persistent. So that means that when we turn the computer off, anything that's stored in this memory gets erased. Now, of course, we sometimes need to store things more long-term and one device on which we can store things long-term, even when the computer is turned off is a hard disk. So data often gets stored on hard disks in the form of spreadsheets. So for example, here's a spreadsheet that shows emissions_data for different countries. Right, I open the spreadsheet in Excel. You've probably all seen this before, right? The spreadsheet has four columns. The name of the country, the particular year, the amount of CO2 emitted by that country in that year, and the amount of methane emitted by that country in that year. And so that spreadsheet might be stored on a hard disk or it might be stored somewhere on a cloud server. That's another common option these days for storing things long-term. If we want to process this data, if we want to analyze this data, if you want to do anything with this data through a program, we first have to, or the program first has has to load this data into memory. And here is one example or one program that does this. So in the rest of the video I'm going to talk about what happens when spreadsheet data is loaded into memory so that a program can process it and analyze it. So what you see here are the first two lines of a program for analyzing the emissions_data. The first line simply associates a filename. Sorry, it simply associates the filename, which is the string emissions.csv, with the name filename. So when the first, when the processor executes this first line of the program, something changes in memory. And specifically what changes is that in the symbol table will have a new name. The name is filename. And associated with this name, filename, will be the string emissions.csv You should be familiar with this from previous videos that you have seen. Right, so this first line of, of, of the code here is just a simple creation of a new name. Now the second line of code, if you look at this, also adds a new name to the symbol table. So the left-hand side of this, of this line of code looks exactly the same pattern as the previous line, right? So it will introduce a new name. The new name will be emissions_data. And then the question is,

what will be the value that will be associated with that name? As you can see in the program, the function read_spreadsheet is called and the name, the filename, so emissions.csv, is passed to that function as a parameter value. The function read_spreadsheet is a function that produces a representation of this spreadsheet data that are stored on the hard disk in memory. So it produces a memory representation of the information that's stored in that spreadsheet file on the hard disk. In many programming languages, this kind of data representation is called a dataframe You can think of a dataframe as representing a table that consists of multiple columns of data. So perfect for representing spreadsheet data In this case, right?, the dataframe has four columns. The country, the year, the CO2 emissions, and the methane emissions. Exactly the same information as is stored in the spreadsheet on the hard. disk. One important detail to notice is that the spreadsheet data on the hard disk and the dataframe in memory, that they are two different things. So the spreadsheet is still stored on the hard disk but now in addition, we have a dataframe in memory which represents the same information. So they represent the same information, but they're two separate representations of the same information. If we, and what that means is if we change one of them, the other one will not be affected. So for example, if I now opened this spreadsheet on the hard disk with Excel and modified the spreadsheet file the spreadsheet file on hard disk would be changed but actually the dataframe, because the dataframe was created before I made that change on the hard disk, so the dataframe would still reflect the old data. And the reverse is also true. If my program makes changes to this dataframe by changing certain values or maybe adding a row at the bottom, the dataframe in memory changes. But the spreadsheet file that is on hard disk does not automatically gets, get modified as well. Another important detail to notice is the way how we represent the association between the name emissions_data and the data frame. So you can see here the way we are visualizing it is by just drawing an arrow that starts in the values column, column of the symbol table and then points at the dataframe. So if you've watched videos about sequences before, you've encountered this kind of arrow before. If you have not watched the videos about the sequences, then this is new for you. And in that case, all you've seen so far is names in the names column of the symbol table that are associated with values that are listed in the values column of the symbol table. And so the filename here's an example of this, right? With dataframes, this kind of representation is hard to do because dataframes represent a whole collection of values, right, they don't represent just a single value. And also dataframes can grow and shrink. So for example, as I just mentioned before already, you know, we might add a

row to the dataframe or we might delete a row. So the processor won't know at the time where this name emissions_data is created and added to the symbol table, the processor doesn't know how much space it will ultimately need in the values column to represent the whole dataframe. And so instead, the dataframe is stored somewhere in memory, and then in the symbol table, the processor just stores a reference to the memory location where the dataframe is actually stored. And we visualize this by drawing an arrow that starts in the values column and points at the dataframe. So in this video you have seen what happens when a program reads data that are stored on a long-term storage device like a hard disk or a server in the cloud and reads it into memory. And you've seen what dataframes to represent tabular data look like when they're associated with a name in memory. In the next video, we're going to explore programs that manipulate dataframes.

Working with Dataframes

So in the previous video you learned what happens when a program reads data from a spreadsheet file into memory. For example, these two lines of code could be the beginning of a program that analyzes the data stored in the spreadsheet file emissions.csv. There. The, when these two lines of code are executed, you end up with two entries in your symbol table. You end up with a name filename, and you end up with a name emissions_data. The name filename is associated with a string, the string emissions.csv, and the name emissions_data is associated with reference to a dataframe. And, as you also learned in the last video, a dataframe is a way of representing tabular data, like for example, data that comes from a spreadsheet. In this video, we're going to introduce some typical ways of how programs interact with dataframes. And we're going to show you how this interaction with the dataframe affects what is stored in memory. We're first going to look at a number of ways to access information in the dataframe, so we just access it, and then we're going to look at ways to modify information in the dataframe. So in this first example, we are accessing a particular column in the dataframe, and we're specifying the column we want by its name. The first two lines are exactly as before. So we have our name, filename, we have the name emissions_data in the memory and they're associated with the same values as before. And then in the third line, we call a function called get_column. So the purpose of this function is to get all the data that is in a particular column in the dataframe. This function

needs two pieces of information. It needs to know which dataframe we want to access, and it needs to know which column we want, we want to access. And then it returns all the data in that column as a sequence of values. So in this example here, we're accessing the dataframe that is associated with the name emissions_data, the first piece of information we're giving, and we want all the data that is in the column that has the name country. So we're giving the name of the column as a string here. So this column contains a sequence of strings, right? So it's all the values that are in this column and those are all strings representing country names. Another thing that this third line of code does is it creates a new entry in the symbol table. In the symbol table, the first part of the line here creates an entry for the name countries. And then this name countries is going to be associated with a sequence of values, and those values are going to correspond to the values taken from the column that has the name country. So in this case, the sequence of values might look something like this. USA, USA, Canada and then so on a list of other country names. Sometimes we may want to access a column not by its name but by its number. So this may be because we are accessing or we are working with a dataset that doesn't have column names. Or there could be some other reason for it. And this example, again, it starts out with the same setup as before. emissions data, filename, same as before, but now we're using a function get_column_by_number. This function also needs two pieces of information, right? We still have to say which is the dataframe that we want to work with. But then instead of giving the name of the column we want to access, we give a number that indicates which number column we want. For example, in this case, if we say the column with a header country is column number 1. column number 2 would be the year column here. And so what we end up with in memory is in the symbol table, we get a name that has years. And this name years is associated with sequence, a sequence of numbers in this case. And these numbers are all the numbers, all the years that appear in this year column in our dataframe. Sometimes what we really are interested in is just a single value in a particular column. So we're ultimately not interested in the whole column, but we want just one value from this, from this column, and that is what this example shows. So here we are getting the column in line number 3. We're getting the column from the emissions_data dataframe. We're getting the column CO2 that has the header CO2. So here's our dataframe column CO2, that gives us a sequence of numbers. And so in our symbol table, we get the name co2_emissions associated with a sequence that in this case might be something like 301000, 298000, and then so on. Now, if we want just an individual item from that, that, that sequence, we can extract individual items. So for

example, here we are saying get_item. This function also needs two pieces of information, right? So we need to know which sequence is it that we accessing and then we need to say which item in the sequence do we want? So here we're saying access the sequence associated with the name co2_emissions. And from that sequence, co2_emissions get item number 2. So memory, Here's the name co2_emissions rate is associated with this sequence here, we want item number 2. So if 301000 is item number 1, item number 2 would be 298000 And so the function get_item turns the value298000 And then the rest of line number 4 this first part here of line number 4. makes sure that in our symbol table, the name co2_us_2001 is added, and this name is associated with a value 298000 In the three examples that we've seen so far, the program accessed columns from a data frame, but we can also do the same thing with rows. So in this example here, we're interested in the first row in the dataframe with a name emissions_data. Right? So we are using, instead of using the function named get_column_by_number, we're using get_row_by_number here. But as before, as with a column, there's two pieces of information we have to provide. We have to say which dataframe are we looking at and which number row do we want here? We want row number 1. And then it's, the rest looks very similar to what we saw with the column. So the information that is in that first row, right? It's a sequence of values. So in this particular case and this dataframe, one row consists of four items, one string, and then 4 numbers. And so the values, the sequence of values that represent the first row is then associated in the symbol table with the name us_2000 in this program. If we are interested, not in the whole row, but again just in one item from that row, one cell. We can do the same thing as we did with, with, with columns, right? So we first get the whole row. And then we can say, okay, from that row, we use the function get_item just as before, right? Because a row is a sequence that all the values from that row I've represented as a sequence, just as the values from a column are represented as a sequence. So we use this function get_item to say from that sequence, us_2000, that it represents the first row of the emissions data from that sequence. Retrieve item number three. So in this case, this is item, USA is item number 1. 2000 is item number two, and then 301000 is item number three. And what this little program snippet does is that it associates the value 301000 with the name co2_us_2000. Finally, if we know both the row and the column number of the value that we're interested in, we can directly access a single cell in the dataframe. Here in this example, we're using the function get_cell and this function needs three pieces of information. We need to specify which dataframe is it that we are interacting with. And then we have to

specify both the row number and the column number. So here we are interested in the value that is in row number 1 and cell number three. So in this example, what that would mean row number one is this row cell number 3 is 1, 2, 3 is in the CO2 column. It's the value 301000 So we would end up with a symbol table that in addition to the filename and the emissions_data, has one additional name, which is co2_us_2000 And the value associated with that name is the number 301000, which is in the third column, first row. Okay, up to now the examples all involved accessing values in a data frame. They did not change a dataframe. Now let's look at a couple of examples where the program modifies the dataframe. We use the same starting point as before. The dataframe from the spreadsheet emissions.csv has been read in and is represented in memory as a dataframe in the form of a dataframe as shown on the slide. Now, we call this function set_cell, which the purpose of this function is to modify one cell in the data frame. This function requires four pieces of information we have to specify. The dataframe that we want to modify, we have to specify which cell you want to modify. We can do that by specifying row number and column number. So here we are saying row number 1, column number 3 is what we want to modify. And then as the fourth piece of information, we have to say what's the new value that should go into this spot? And when we do that, you can see that row number 1, column number 3, the value changes to this new value, right? So if I go back, you can see this is the value that was in there before, 301000. After this function set cell is executed, the value in that's what is now 303000 right? Before, after the change. At this point, it's important to remember that the spreadsheet from which the dataframe was originally created, does not change when we make a change to the dataframe, right on the hard disk in the spreadsheet, the value is still 301000, but in-memory in the dataframe, the corresponding cell has been changed. So the final example here, oh here, here's the picture to go with this. So on the hard disk, the spreadsheet has not changed. It's still 301000 in-memory. In the dataframe, the value has changed to 303000. So in this final example, we're adding a whole row to a dataframe. So line number 3 here specifies a row and new entry. I made a new row for our dataframe, Rows in this dataframe are four items long. So this sequence here has four items in it for values. And we add and then call the function add_row to add that row to the dataframe. So this function requires just two pieces of information. We have to say which dataframe we want to add this row to And then we have to say, what is the row that we want to add what is the sequence of values we want to add. And when we do that, again, you can see how the dataframe in memory changes. This is what it looks like before. After we have called add_row there's now an

additional row at the bottom that has the new the new values in it. And as before though, on the hard disk, the spreadsheet is not changing. Just the dataframe in memory is changing. If you did want to save this new version of the dataframe in a spreadsheet file on the hard disk so that you could work with it later on, you would have to explicitly tell your program, but that's what you want to do. You would have to call another function that takes as a dataframe and saves that dataframe to the hard disk. So this would kind of be a function that is the sort of the flip side to the read_spreadsheet function, right? So instead of reading a spreadsheet from the file, we would need a function, write_spreadsheet or so that would take a dataframe and write it to a spreadsheet file. So that is if you wanted to keep the modifications that you made to the dataframe for later, for later processing.

Conditional selection of rows from dataframe

In a previous video, you have seen how a program that wants to process some data that's saved in a spreadsheet on the hard disk first needs to read that data into memory, and in memory, then, this data is represented as a dataframe. So for example, given this little two line program here that reads in some data, we would end up with the symbol table that has two names in it, filename and emissions_data. And emissions_data would be associated with this dataframe that is created from the data that is saved in the spreadsheet. In another video, you've then seen how to access the data in the dataframe. And in particular, you have seen how we can retrieve a single row from the dataframe by asking for, for example, row number 7. So, you know, asking for a particular number row. In this video we are going to look at another way of selecting certain rows from the dataframe. In this case, we're not just selecting a single row. But what we're doing in this video is that we're going to look at how to select a whole subset of rows that satisfy some condition. So for example, let's assume we are working with our emissions_data, which has emissions for different countries for many different years. And we want to drill down and we want to further analyze the emissions of the US over time. So really from our, from all the data that's in our dataframe, the only rows of the dataframe that we're interested in for this particular analysis are the rows which have USA as the country name, right? So we would be interested in this first row here, but then the next row has China as the country name. So for the particular analysis where we're looking just at the US emissions, we are not interested in that row. Then this last row that's shown here,

we would be interested in that again. So here is how we might do that in, in a program. So if you look at the three lines of code here, the first two lines are the same as before, and then we have this third line of code here, which looks quite involved. So let's break this down a little bit. So what we have is we're calling a function called get_rows_by_condition. The function has two arguments. The first one, emissions_data, so with that first piece of information that we're providing to this function, is just which dataframe we are trying to access. And then the second parameter that we are giving here is this, this more complicated expression. And that's what we're going to look at now in a bit more detail. So you can see that as part of this expression we have a function call that you've seen before get_column. So that also showed up in the video that talked about how to access data in a dataframe. And so what do you know from that video is that what this function does is it retrieves a single column from the data frame. In order to be able to do it, what we have to do is we have to specify which dataframe we're accessing, in this case, emissions_data. And we have to specify what name, what the name is of the column that we're interested in. So in this case we're interested in the country column. And so what that gives us then is a sequence of strings. where each string represents a country name. So that sequence of strings might look something like this. We have USA, then we have China. You know, some other things. USA appears again, some other things. So it's a sequence of country names. Now, the second part of this expression is == and then string USA. So this == operator is an operator to tell the computer to check for equality. So what we're trying to say here -- is this first part, so what ever is the result of this function call, is that equal to the string USA? Now, we just, we just talked about how the first part or the result of this function call is a sequence of strings. So if we were to take this literally and we'd say, well is this sequence of country names that comes back from, from this function call, is that equal to the single string USA? The answer would of course be no, because one is a sequence of strings and the other one is just a single string. Most programming languages though that deal with data, are made for dealing with data, they're smart enough to understand that if I have a sequence of strings and I'm asking, is this == so is this equal to a single string? What I'm really asking for is. What I'm really asking for is a comparison where we are comparing every individual item in the sequence to the string USA. So in this case, what that means is we are going through our sequence of country names taken from the column, the country column. And we check whether they are equal to the string USA. So we have USA, is that equal to USA? Yes, that's true. China. Is that equal to USA? No, that's false, right? We

do. And then we do the same thing for all the other entries in our country column. So for each one of these entries we get true or false. Now, we can combine that with our or we can apply this condition to our dataframe emissions_data in the following way by basically saying, okay, now we have these true and false values we have as many true and false values as there are columns. Not sorry, yes, they are rows in this dataframe because essentially what we did is we went through the country column and for every entry in the country column, we said, is this equal to USA true? Is this equal to USA false, right? And so now what this function get_rows_by_condition does is that it goes through and it just selects, it retrieves all the rows for which we said that this comparison here equals true, right? So USA is equal to USA. So the condition is true, which means this row would be selected. The next row. China does not equal USA. So the condition is false. This row would not be selected, and so on. So what that would give us is another dataframe, a new dataframe. But this new dataframe only contains a subset of the data in the original dataframe. Specifically in this case, it contains all the rows where the country name is USA. And then with that new smaller dataframe, we can then do our analysis of emissions in the US. So what we get in memory in the end, it looks like this. We have a symbol table. In the symbol table we have three names, finally, emissions_data, and us_emissions. emissions_data is associated with a reference to a dataframe which contains all of the data from the original spreadsheet. And us_emissions is associated with a reference to another dataframe. And that dataframe contains only those rows where the country name is USA. Now, here is another example. For this example, let's assume that we want to look at the recent developments in greenhouse gas emissions. So we can again use this function, get_rows_by_condition to extract out the rows that we're interested in. And let's assume by recent emissions we mean everything since 2010. So we want to ignore, or for this particular analysis, we want to ignore a greenhouse gas emissions that happened before 2010. And we only want to look at the emissions since 2010. So the condition that we are writing here, instead of looking at the, the column, the country column, because which we did the last time when we selected based on country names, this time of course, we have to look at a different column. And the column we're looking at is the year column because we want to select based on the year in which the emissions happened. So then the second part of this condition expression here, we are saying that we want emissions. We want rows where the year, the number in this year column is greater or equal to 2010. So if we were interested in only in the emissions of exactly the year 2010, we would use our == operator here again to get only those rows where the

year in that country column is exactly equal to 2010. But we are interested in all emissions since 2010. So we are interested in all rows where the year in the country column is equal to 2010 or greater than 2010. And this operator here, >= does that for us. So in this case, what we would end up with then is a symbol table in which again we have three names, filename and emissions_data are just as before. And then the last name emissions_since_2010, which was introduced by this third row in our program here, would be associated with a reference to an dataframe, new dataframe. And this dataframe would contain all the rows from the original dataframe, where the number in the year column is equal to 2010 or greater than 2010. And so this is how you can extract a set, a subset of rows from a dataframe to create a new smaller dataframe if what you want to do is a targeted analysis that looks at just a subset of the rows in your data frame.