Intro to Computer Science

Lesson 02 Notes

How to Repeat

Introduction

Welcome back for Unit 2! I hope everyone is getting confidence in the things you learned in unit 1. We're going to continue to build on those in this unit as well as in rest of the course. Our main goal for this unit is to make the web crawler, instead of just finding one link in the page, to find all the links on a page, so that we can follow these links and collect more and more pages on the web. To do that we need two big new concepts in computer science. The first is procedures. Procedures are a way to package code so that we can reuse it more easily. The second is control. We need a way to be able to make decisions, and to do repetition, to find all those links on a page. What we saw at the end of Unit 1 was a way to extract the first URL from a web page-- that's great--we could find the first target. But if we want to build a good crawler, we don't just care about the first one, we care about all of the links on the page. We need to extract all of those links-- figure out where all of those links point to-- so we'll find many more pages to crawl than just the first one. So, that's the goal for this class as far as building a web browser. To do that, we're going to learn two really big ideas in computer science. The first is about procedures, and that's a way to package up code so we can use it in a much more useful way than we could before. The second is about control. Control structures will give us a way to keep going to find all the links on a page.

So, let's remember the code we had at the end of Unit 1. We solved this problem of extracting the first URL from the page-- we assumed the page was initialized to the contents of some web page. We initialized the variable "start_link" to the result of invoking "find" on "page," passing in the start of the link tag. Then, we initialized the variable "start_quote" to the result of finding, in the page, the first quote following that link tag. Then, we initialized the variable "end_quote" to the result of invoking "find" on "page," to find the first quote following the start quote. And, then, we assigned to the variable "url"-- extracting from the page-- from the character after the "start_quote," to the character just before the "end_quote." Then we could print out that URL. This worked to find the first URL on the page. If we wanted to find the second one, we could do it all again. We could say, now we want to advance so we're only looking at the rest of the page. We could do that by updating the variable "page," assigning to it the result of the rest of the page, starting from "end_quote"-- and,

remember, when there's a blank after the colon that means select from this position, to the very end-- and then we can do all the same stuff. We'll do "start_link" again; ... we'll do "start_quote" again; ... Now we've got code that's going to print out the first URL-- keep going, updating the variable "page"-- and then doing the exact same thing-- printing out the second URL. If we wanted to print out the first three, we could do it again...

So now, we've got code to print out the first three URL's on the page-- let's scroll all the way up--so you've got-- print out the first one--keep going-- print out the second one--keep going-- so this can go on forever. The reason we have computers is to save humans from doing lots of tedious work. We don't want to make humans do lots of tedious work-- certainly, typing this out, over and over again, would be really tedious, and it wouldn't really even work that well. We have pages with hundreds of links, but there are other pages with only one or two links. So, it wouldn't make sense to copy this hundreds of times. There's always going to be some web page that has more links than we have copies of it-- and any page that has fewer copies, we're going to run into problems because we're not going to be finding any of those links. So, our goal today is to solve all of those problems.

Motivating Procedures

The first thing we want to do, is what's called "procedural abstraction." This is a really powerful idea. One of the ways we avoid having to do things over and over again by hand, is that we can write the code-- write it once-- and use it many times, changing the inputs to get different behaviors. If you look at each segment of code here, these lines were copied exactly the same way. We did the same thing, over and over again-- the same five lines of code-- to find the start of a link in "page," to find the start and end quote and then to extract the URL, and then to print the URL. The only thing that was different was the value of "page." We kept updating the value of "page"-- we did that as we went through the code-- we updated the value of "page" with this assignment and, before this one, we updated the value of "page" with this assignment. So, the idea behind procedural abstraction, is that anything that we're doing over and over again we want to abstract-- we want to make it a procedure-- and we want to make the things that change, inputs. That means, instead of having to type in this code over and over again, we want to make "page" an input-- "page" is the input; that's something that changes-- and we want to make the URL-- that's the result we want-- an output. That's our goal, to turn this into a procedure, where "page" is the input-- we can do the same code, over and over again, and every time we do it, we want to get this URL value as the output.

Introducing Procedures

The way to do that is what we call a "procedure." A procedure is something that takes inputs in-- there can be more than one-- does some work on those inputs; and produces outputs as results. The idea of a procedure is a very powerful idea. This allows us to use a small amount of code to do many different

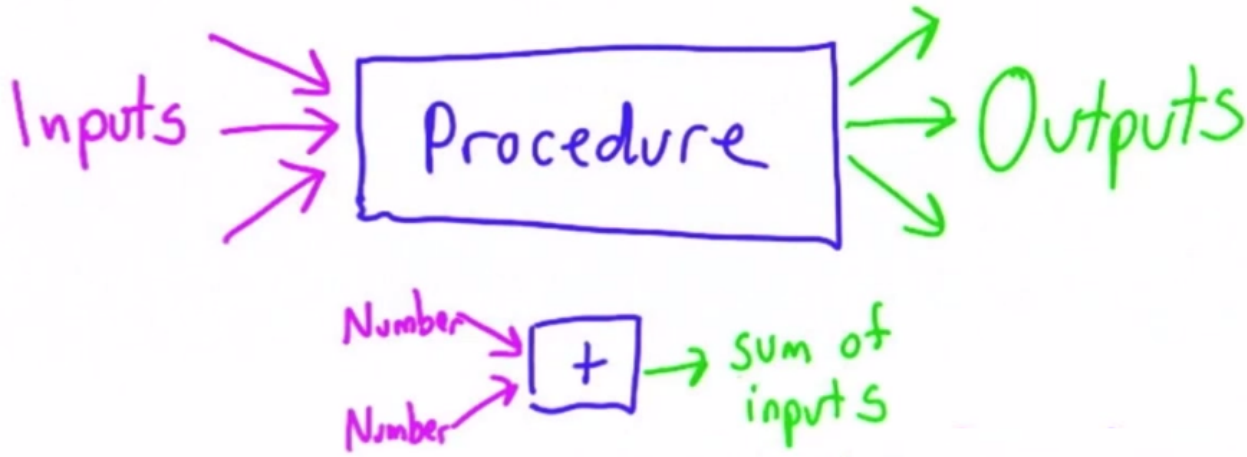
things. We can have the same code operate on different inputs. Whatever we pass in as inputs to the procedure will be what that code operates on-- and it can do different things depending on those inputs-- and it produces outputs that tell us the result, based on the inputs that we passed in. You've seen things very similar to procedures in the previous unit. You saw the built-in operators-- operators like "+." "+" took two numbers as its inputs, and, as its output, it produced the sum of those two numbers. Because it's a built-in operator, it's a little different from a procedure-- the syntax that we use for it is different because it's a built-in operator-- but, in terms of being something that abstractly operates on different inputs and produces the corresponding output, it's very much the same idea. What we're going to learn in this unit, is how to make our own procedures. Python provides a construct for doing that, and the grammar is to have the keyword "def"-- that's short for "define"-- followed by a name. And the name of a procedure is just like the name of a variable. It can be any string that starts with a letter, can be followed by letters and numbers and underscores. Anything that we could use as a variable, we can use as the name of a procedure. Then there's a left paren "(" the left paren is very important-- that's what makes it a procedure-- and after the left paren we have a list of parameters. "Parameters" is just a fancy name for the inputs to the procedure. After that, there's a colon ":" This is what says we're making a procedure-- it has this name; it takes these inputs. Then what we have to do is actually define the code.

So, what follows that is a "block"-- and a "block" is just a sequence of statements, the code that we want to run, as the body of the procedure. What the inputs are is just a list of names, separated by commas. We can have as many inputs as we want. There could be no inputs. In that case, what the parameters would look like is just two parens with nothing between them - "()" There could be one input. In that case, we'd have a paren, followed by a name, followed by a closed paren. Or there could be many inputs; in which case, we'd have a paren, followed by the name of the first input, followed by the name of the second input, followed by the name of the third, followed by as many inputs as we want. So, we could have five inputs, give them the names "a," "b," "c," "d," "e." This is not a good idea, usually, to give our parameters meaningless names; that makes it hard to remember what they are when we use them in the code that we write as the "block." We're much better off having parameters with names like "page" that remind us what they mean. The body of the procedure is the "block." This is the code that runs to execute the procedure. The "block" is indented inside the definition. Usually we like to use four spaces-- that's just the convention in Python, and that makes it easy for people to see and read the code-- but all the lines in the "block" have to be indented the same amount. That's how the interpreter knows that we've got to the end of the procedure, when we get to some code that's not indented anymore.

Quiz: Procedure Code

So, let's go back to the code we have for extracting the first URL in a page and see how we can turn that into a procedure. And remember, we made many copies of the same code if we wanted to extract many links from the page. What we want to do now is take the part that we kept having to copy and

figure out how to turn that into a procedure. What we want to do is take this code-- we're going to take all the code here-- I'm leaving this part out of the procedure, because that's what we want to do with the results-- what we want to do is take all this code and turn it into a procedure. That means, instead of having to do all this work each time, what we want is a procedure that will start from what we have here-- we have the "page." What we want to do is from the "page," do a bunch of work; get the URL. Then, we want to do some stuff with the URL, want to do some stuff with the page, and then we're going to do it all over again. Our goal is to figure out how to turn this part of the code into a procedure, so we could keep reusing that, each time, instead of writing the same code. We need a little more room, so let's shrink the code we had.



Now the question is, how are we going to make that procedure? Let's start with a quiz: When we think about making a procedure, the first thing we always want to do is make sure that we understand this: we need to know what the inputs are to that procedure; we need to know what the outputs are. If we don't understand the inputs and outputs, then there's no hope of writing the procedure correctly since the main purpose of the procedure is to map the inputs into the outputs. Let's see if you can figure out what the inputs should be for the procedure that we want to make to replace this code. I'm going to give the procedure a name. The goal of the procedure is to find the URL that's the next link target in this page, so we'll call the procedure, "get_next_target"-- it's important to give good names to procedures; that's how we remember what they're supposed to do-- and we'll use the name "get_next_target." So, our question now is, in defining this procedure, what should its inputs be? I'll give you some choices: its input could be a number giving the position of the start of the next link; its input could be a number giving the position of the start of the next quote; its input could be a string giving the contents of the rest of the web page; and, for the final choice, we could have two inputs-- the first, a number giving the position of the start of the next link, and the second, a string giving the contents of the page.

[ ] Number giving position of start of link
[ ] Number giving position of start of next quote
[x] String giving contents of the rest of the web page

[ ] Number giving position of start of link, string giving page amounts

Answer:

The answer is that the input should be a string giving the contents of the rest of the web page. That's how "get_next_target" works. It finds the first target from the current page. One way to see that, is to look at the code we're trying to replace. This code defines lots of variables. It gives them values. The only one it uses on the right side, before giving it a value, is "page" When it uses "start_link" on the right side here, well the code defines "start_link" on the line above, so, it didn't care about the value of "start_link" going into this code. The only variable whose value we care about going into this code is the value of "page," and what "page" is, is a string giving the contents of the rest of the web page.

Quiz: Output

So, that's the input. Now, let's think about the output. Again, we're looking at the same segment of code that we're trying to replace with a procedure. What do we think the output should be? This is going to be trickier. We need to think about what we need to know after the code runs. We don't have access to these variables if we're making this a procedure. We need to get--as outputs--everything that we want to use after the procedure. So, the question is, what should be the outputs of "get_next_target," the procedure that we're going to define to replace this code. The first choice is the output should be a string giving the value of the next target URL found in the page. And that's what the variable "url" holds here. The second choice is we should output both the URL and the page. The third choice is that we should output both the URL and the value of "end_quote," the position where the end of the quote is. And the fourth choice is to output the value of "url," as well as the value of "start_link." That's the position where we find the beginning of the link.

[ ] String giving the value of the next target url (url)
[ ] Url, page
[x] Url, end_quote
[ ] Url, start_link

Answer:

So this is a little bit trickier. To answer this one we have to look at the code after the procedure, and remember we said the procedure is replacing these four lines. After the procedure what we do is print the value of "url." So that means we definitely need at least "url" as our output and all of the choices here included "url." But that's not all we do, so we have to look at the second line as well. This uses both "page" and "end_quote." The value of "page" here is the same value of "page" before the procedure. So we don't actually need to return the value of "page." It doesn't need to be an output from the procedure. We already know that. We knew that before we called the procedure. The procedure doesn't change the value of "page." No sense making it an output. So this answer doesn't

make sense. We don't need "page." It does make sense to make "end_quote" one of the outputs. The reason we want "end_quote" as an output is we need to know where the end of the quote was to advance the page, so the next time we look for the next target, we don't find the same one that we just found. So we actually need this, so this is the best answer to the question, is that we need two outputs, we need both the url, and the value of "end_quote." The fourth possibility could work. If we return both the url and the value of "start_link," We could figure out a way to advance the page to not find the same quote, but we'd basically have to redo all this code. We need to look for the next quote, starting from "start_link." We need to look for the closing quote for that. Basically we'd need to compute the value of "end_quote" again. So it's a lot more useful to return the value of "end_quote," than to return the value of "start_link." That's why the third choice is the best choice.

Quiz: Return Statement

So now that we know the inputs and the outputs, the procedure almost writes itself. Especially since we already know the code that we wanted to execute before we defined the procedure. So I'm going to write the procedure, by modifying the code here. So first what we're going to do is make a little space. Erasing the code that's not part of the 4 lines that we want to turn in the procedure. So now I haven't modified the code at all. What I have is, almost exactly what I need for the body of the procedure. This is what we want to do. What we need to do is modify that to get in the inputs, and turn it into a procedure.

So, to make it a procedure, we need to use the def name syntax. So we're going to define the procedure "get_next_target." So now we need our parameters, the parameters are the inputs for the procedures and we decided what the input should be. Is the web contents that's the string giving the contents to the webpage and that's what we had in the variable page. So we can call that page again here, that's what we are passing in as our input and now we have the body of the procedure. The code is exactly the same as the code we had before. But this time, instead of page being whatever it was here, page is whatever we pass in as the input to the procedure. And we can change the name of the page. It sort of makes sense to change that name. This code works no matter what we pass in. It doesn't have to be a web page. Any string that we pass in, this code will find it. So it makes more sense to give it a slightly more generic name. We'll change the name to "S." If we change the name of the parameter, well everywhere that we use page, now we don't have the value of page. What we have is the value of S, which is the name of the parameter.

So we'll change all of those to S's as well. So we're almost done. There is one big thing we have left to do and that's how do we get the outputs. Right we said what we wanted the outputs to be is the URL and the value of end-quote. We need some way of getting those back and the way that you do that is to use return. Return is a special construct in python. So we have the keyword return, followed by a list of all the things that we want to return from the procedure. So what we return is a list of any number of expressions, separated by commas. We can have zero return expressions, that would mean

there's no output. And it actually is useful to have procedures with no output sometimes. That's not true if we just think of procedures as things that map inputs to outputs, but procedures can also do other things. They can do what we call side-effects, and what side-effects are, are things that we can see, but that aren't the outputs. And an example of a side-effect would be, well, if we printed something out, we would see that result happen. We'd see everything that happens when the code and procedure runs. But we wouldn't get it as an output.

So, in this case, we do want to have outputs from get next target. We want to return results, so we know how to continue. And we decided what the output should be, are the value of the URL. So we want to find the string that's the next target. And the value of the end quote, so we know the position where it was found. So let's see if you can figure out how to finish the return statement. And your goal is to figure out the code that we need after the return, to finish this procedure.

Answer:

The answer is we need to return two things. We want the outputs to be the value of "url" and the value of "end_quote" so we know both the "string" that was found ”that's the target of the link ”as well as the value of "end_quote." We can do that by just returning those two values. We're going to have returning "url," followed by a comma, followed by "end_quote." That means that the two outputs from our procedure are the two values of "url" and the value of "end_quote." [Drawing smiley face.] Congratulations! You've defined your first procedure. This deserves a big smiley face. Procedures are really important ideas in computer science. We're going to find lots more procedures both today and through the rest of the course. Almost everything we do "in terms of building programs" is about building procedures. Those are the main things that we use to break down problems into things that we can solve. We're going to have some more quizzes about procedures soon, but first I want to give you an idea of how rich inputs and outputs can be when we do computations. We'll do that by looking at what happens when you build a self-driving car.

Dave, Sebastian, and Junior


DE: So, I'm here with Sebastian, and Junior, one of Sebastian's self driving cars. Sebastian is going to give an idea of how rich inputs and outputs can be by talking about the kinds of inputs and outputs that he has on the self-driving car.

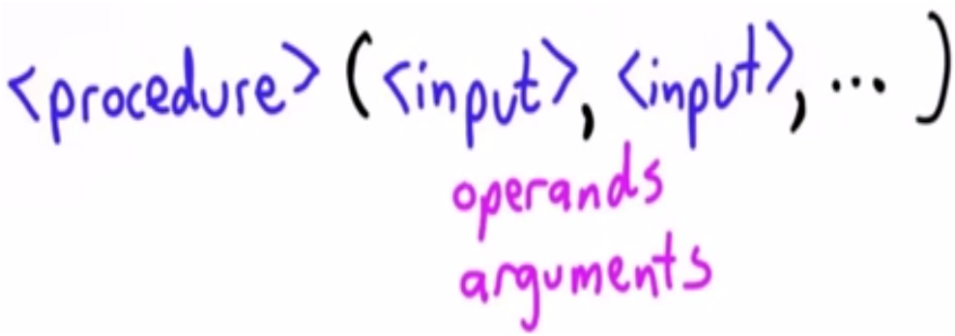
ST: So, taking data in is essential to making self-driving cars work. This machine has a lot of sensors that take data in at enormous rates. There is a laser range finder on the roof that rotates. And it takes scans 10 times a second of the entire surroundings or range scans. It has a security camera, very much like Google Street who's able to see in all directions. In fact, this camera played an important role in Google Street's history. And then it has stereo camera system, and also has a GPS system that we use to localize the car course [inaudible] based on satellite navigation. So, all this data, which are millions and millions of data points per second, get fed into the computer system that sits in the robot's trunk. So, this is the data. This is the laser data, but a million points per second. And you can see in this 3-dimensional depiction, the robot is parked in the garage, it sees the back wall, and it can see all kinds of other things around it. It uses that exact same data to avoid collisions with other cars, bicycles, pedestrians, and so on. And finally, the data gets funneled back to the car. These are the outputs of the computer system. And they go into the car: four little switches over here. They actuate the steering wheel; they use the brake, the gas. You can't really see it? It looks like a regular car; and it is. But if I hit this one button over here, it turns into a self-driving car.

Using Procedures

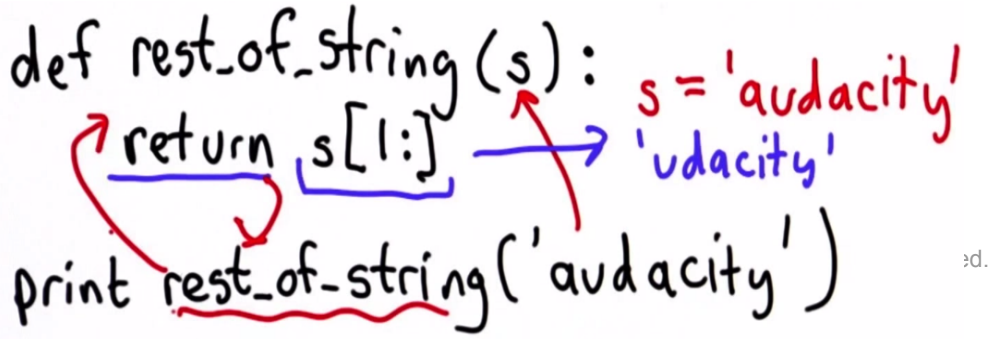
Unfortunately, we're not quite ready to be all smiley. Sebastian tells me it's going to be a few years before I can get my own self-driving car, but the bigger problem is we haven't yet talked about how to actually use the procedure. All we've done is make them; until we can actually use them we don't have a good reason to be really happy yet. We're going to learn how to use them next, then we'll be back to being smiley. Now we are going to learn how to use a procedure. The way to use a procedure we need the name of the procedure, followed by a left paren, followed by a list of inputs. There could be any number of inputs, but it has to match the number of inputs the procedure expects. These inputs are sometimes called "operands." They are sometimes called "arguments." We're not going to argue about that. We're just going to call them inputs. You have actually already done something quite similar to this.

Back in Unit 1, you learned about using "find" on "strings." With "find" you would pass in one or two inputs. The first input was a "string." That was the string you are looking for that's the first input and the second input could be a number ``the position where you start looking for that string. We use "find" in many ways in Unit 1, as well as you used it yourself in the homework for Unit 1. "Find" is a little different from the procedures that you define yourself. First of all, it's built in. The other thing that was different is that, instead of just having "find," we had another input that was really over here. We have the string that ``what's really another input we're doing th

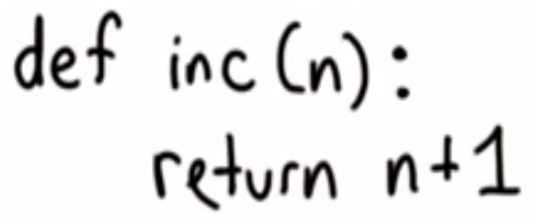
We'll talk in a later class about why that's done differently, but there's another case where one of the things is a little different from that and we won't get into that in this course but in a later course you'll learn more about what this really means. For all the procedures that you define yourself, we won't have any object to invoke them on. We'll just have the procedure to call and the arguments or operands or inputs as you like to call them to pass in. Let's see how that works with a simple procedure. I am going to define the procedure "rest_of_string," and we'll give it the parameter "s," so that means it takes one input and we are going to use the name "s" to refer to the value of that inp to the end. We will use the "string" index with the first letter removed, so all "strin what we return. So, the output of "rest_ in the input "string." Here's an example say "print rest_of_string." That's our pro going to pass in an input. There's one pa it should be a "string." We'll pass in the s this, execution will jump into the body o now. Instead of running the code here, the interpreter will move. When we call a procedure, it will jump to run the code values passed in as the inputs. We can think of this as there being an assignment that says now the value of "s" is the value of this input that was passed in. Now we are going to evaluate the body of the procedure. In this case there's only one statement; it's this return statement. We are going to find this value, s [1:]. The result of that is going to be the string 'udacity'. Then we got to the return. What return means is we're going to jump back. We're jumping back to where we called the procedure, but now we actually have a result. When we jump back, the value that this evaluates to is whatever value we returned. In this case, it's the string 'udacity.' So we don't have our self driving car, but now you can define and use procedures. This is a really powerful concept. Anything that we are going to do in the rest of the course and anything almost anyone does in programming computers is all about defining procedures and using procedures. Now we should have a big smile. We can think of our procedures in terms of mapping inputs to outputs. We can think of us humans as also mapping inputs to outputs. We have inputs coming in through the eyes, through the mouth maybe we even have a nose. I won't try to draw any of the outputs of our human procedure, but since procedures are such an important concept, we are going to have several quizzes now to check that you understand them well.



Quiz: Inc Procedure



So, for our first question about procedures, The question is, "What does the 'inc' procedure defined below do?" And here's the code for defining 'inc.' We're going to define the procedure that takes one input; we'll call that input "n," and we are going to have our return statement where the expression following the return is "n+1." So what does this procedure do? The choices are: Nothing. It takes a number as its input, and outputs that number plus one. It takes a number as input, and outputs the same number. Or--the final choice--it takes two numbers as inputs, and outputs the sum of those two numbers.



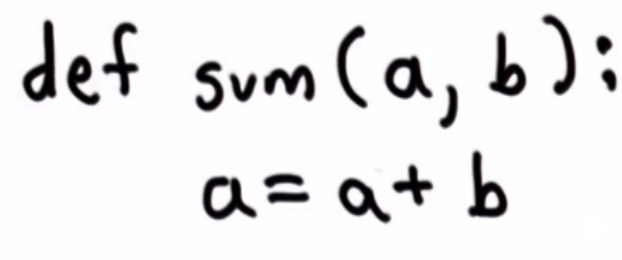
[ ] Nothing
[x] Takes a number as input, and outputs that number plus one
[ ] Takes a number as input, and outputs the same number
[ ] Takes two numbers as inputs, and outputs their sum

Answer:

So the answer is the second choice. The reason--for this we can see that it takes one input-- there's one parameter between the parentheses-- and it calls that input "n," and then what it outputs is-- after the return, it outputs the result of "n+1." So whatever was the value of "n," which is the input, the result of "inc" is the value, plus one.

Quiz: Sum Procedure

So let's try another quiz. This one's going to be quite a bit trickier. Here the question is, "What does the 'sum' procedure do?" And here's the definition of 'sum,' so here we're defining the procedure of sum. It takes two inputs, "a" and "b," and its body assigns to "a" the value of "a" plus "b." So the choices are: Nothing; that it takes two numbers as its inputs, and outputs the sum of the two numbers; that it takes two strings as its inputs, and outputs the concatenation of those two strings. So the final choice is that it takes two numbers as its inputs, and changes the value of the first input to be the sum of the two numbers. This is a quite tricky question. I would encourage you to try some experiments in the Python interpreter to see if you can determine for sure what the sum procedure does.



[x] Nothing
[ ] Takes two numbers as its inputs, and outputs their sum
[ ] Takes two strings as its inputs, and outputs the concatenation of the two stings

[ ] Take two numbers as its inputs, and changes the value of the first input to be the sum of the two numbers

Answer:

So the answer is, it really does nothing. It'll make the computer do some work when we call sum, but there's no good reason to ever call this procedure. It doesn't have a return, so it doesn't produce any output, and it doesn't actually modify the value of the first input in any way that's visible to the caller. It does modify the value of a inside sum, but the caller can't actually see that. Let's see that in the Python interpreter so we make sure we understand what's going on here. We'll define sum to take those 2 inputs and have the body that assigns to a, the value a + b, and we can try printing the result of calling sum. Let's see if it can add 1 + 1. When we run this, what we'll see is the result is actually None. The reason the result is None is, the sum procedure doesn't actually return anything, so there's no value here. Python uses the special value "None" to mean there's no value. So that's not very useful. We could pass in bigger numbers. Whatever we pass in, will still get the result None because sum never returns anything. To make sure it's clear what's going on here, we're going to add some print statements to sum. Printing things out is actually a perfectly good and perfectly useful way to test your code to make sure that you understand what's going on. It's always a good idea to add print statements. Sometimes we call this debugging. If the code has problems, adding print statements to see more what's going on is a good way to figure that out. Here I'm going to print that I've entered sum, and I'll print the value of a, and I'll do the same thing again.

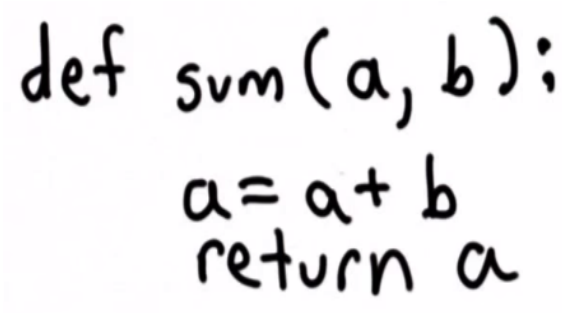
Now when we run it, we see that we "enter sum!" When we enter sum, the value of a is 2. That was the value passed in as the first parameter. The value of b is 123. That was what we passed in at the second input. When we print out the first print, a is 2. We see a is 2. We do the arithmetic. That changes the value of a. So now running the same print, it prints out the value of a as 125. Then we return, we have the result. The result is None. There's no output from sum, and that's what gets printed out.
So if we add a return statement to sum, now we have the original code that we had. I've removed the print statements. So we don't need to see those again. We have this, and now we've added a return statement. Now when we run this, we do get the value 125 because the output of sum is what's returned here, which is the result of a at this point, which is 2 + 123. The other thing I want to make clear without the return is that if we passed in variables for this, it still does not change their value. Let's say we had a variable--I'll even use a. So I'm going to use the variable a. We'll initialize it to 2. We'll have the variable b. We'll initialize that to 123. We'll call sum. We still don't have the return, so it will return no value.

Let's see if the value of a has changed at the end, and it hasn't, and the reason it hasn't is what gets passed in here-- even when we have a complex expression, even when we use a name, or we have some calculation that produces the value-- what gets passed in is the value that that evaluates to. So the fact that it was a name doesn't matter. When we call sum, what's getting passed in as the value of a is the value that a refers to, which is the number 2. We don't actually modify what the variable is.

That's a totally separate variable. Let me draw what that looks like to make it clear what's going on. Here we have our code. So that introduced a name "a," and it refers to the value 2, and it introduced a name "b," and that refers to the value 123. Both of those are numbers. And we have our procedure that we defined like this. It takes the 2 inputs. It changes the the value of the input named a. We're going to call sum, passing in a and b. What happens when we call sum--we have these names here. The names are our parameters. We're going to have a name inside sum that's a. That is going to refer to whatever value is passed in here. So to know what the value is, we evaluate the name a. We get the value 2. This a will refer to the value 2. b will refer to the value passed in as b. It's going to be the number 123. And then when we run sum, we do the assignment. That's going to produce the new number adding 2 + 123. We get the value 125, and then we do the assignment. The assignment changes what a refers to.

Now instead of referring to the number 2, the a that corresponds to the parameter to the procedure sum, now refers to the number 125. Note that the original value of a, the one that we named here, still refers to 2. When we come back from the procedure, so once we get to here, we return. We're back to running the code here. All of the parameters that were visible inside the body of sum, those are no longer visible. They were there for that execution, but they're done. We can't use those names anymore. They were only visible during that execution of the procedure. Now at this point, a refers to the number 2, just like it did before.

Quiz: Sum Procedure With A Return Statement



For the sum procedure to be useful and not do nothing, we need to change it. We need to add a return statement so we actually get an output. So add the return statement, return a, and now we'll re-ask the same quiz. I'm going to erase the previous answer because the answer is no longer nothing. I guess I've hinted at that. You can cross this one out from your consideration, but see which of the others are correct, and I should note it's still a tricky question. More than one of these is correct.

[x] Takes two numbers as its inputs, and outputs their sum
[ ] Takes two strings as its inputs, and outputs the concatenation of the two strings
[ ] Takes two numbers as its inputs and changes the value of the first input to be the sum of the two numbers

Answer:

Now the answer is that it actually does do something. It produces an output. If it takes numbers as inputs then it produces their sum. But it works also if we pass in strings. It would produce the concatenation of the 2 strings, because the "+" operator works on both strings and numbers. Given the name sum, it makes more sense to use it to add numbers, but it will work perfectly well using strings as well. And let me just show you that in the Python interpreter. So now, I changed sum to return the value of a. I have 2 variables that are strings. We've introduced s to 'Hello ' with a space at the end and t to 'Dave!' Now I'm calling sum, passing in s and t, and what I get as a result is 'Hello Dave!' Note that it does not change the value of s. s is passed into the first input. Even though we change the value of a here, that does not change the value of s.

Programming Quiz: Square

Now we're ready for a quiz where you're going to define a procedure yourself. Your goal is to define a procedure named square that takes 1 number as its input and outputs the square of that number. What I mean by square is the result of multiplying the input number by itself. See if you can define a procedure that does that. If it's correct, you should get results like this. So if you do print the result of square(5), the output that's printed should be 25.

Answer:

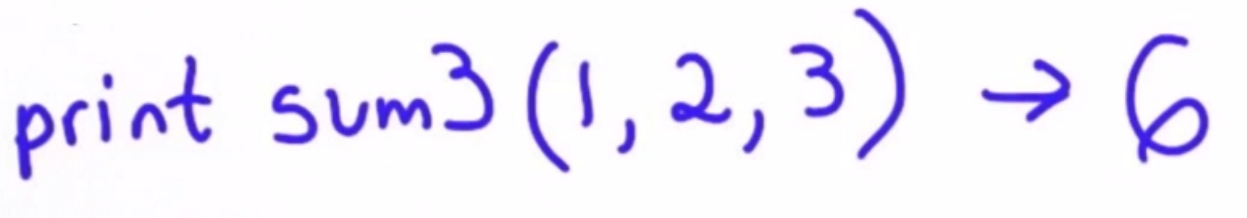
Here's the answer. We're going to define a procedure, so we use the def followed by the name. We'll use the name square, followed by the left parenthesis, and now we need to decide what the parameters are. And remember when we define procedures, we should always be thinking about what the inputs are and what the outputs are. In this case, the input is one number. We see in our example, it was the number 5. We can give that any name we want. N is usually a good name for a number. So we'll use (n) and then we have the colon, and now we're ready to define the body of square. What we want square to compute is the square of the number, so we want to multiply n by n, and that's the output. We can do everything with 1 statement. It's going to be a return statement where the expression that we're returning is n times n. We want to get the value of n squared, and that's what we return. Here's how that looks in the interpreter. We're defining square. Our body returned n * n, and we're going to print the result of square (5). We get 25. We can use a variable. Let's say, x has the value 37. We can print the result of square (x). We get a bigger number that's 37 x 37, and we could also do this.

Let's say we assign y the result of square (x), and then let's print the result of squaring y. That will give us 37 squared, and then we're going to square it again. That gets quite a large number--1.8 million or so. We could do those compositions without variables. We could do this directly. We could print the square of the square of (x), so we're calling square twice here. The first time we're passing in 37, then we're getting the output of that, and then we're passing it into square again. This is called procedure

composition. This is the way most programs are written, that we're going to write little procedures. Each procedure is going to take some inputs and produce some output, But then we're going to use the outputs that we get the first time we use a procedure as the inputs to the next procedure. In this case, we only have 1 procedure defined, so I've used square twice. When we run this, we see the value of 37 squared, which is quite a big number. We'll see lots more examples of how we can compose procedures using the outputs of 1 procedure as the inputs to the next procedure to get lots more interesting things done, and you'll write some procedures that do that yourself pretty soon.

Programming Quiz: Sum Of Three

For our second quiz where you have to define your own procedure, we'll make things a little more complicated. Now you're going to define a procedure. We'll call it sum3, and the reason I'm calling it sum3 is because it takes 3 inputs.



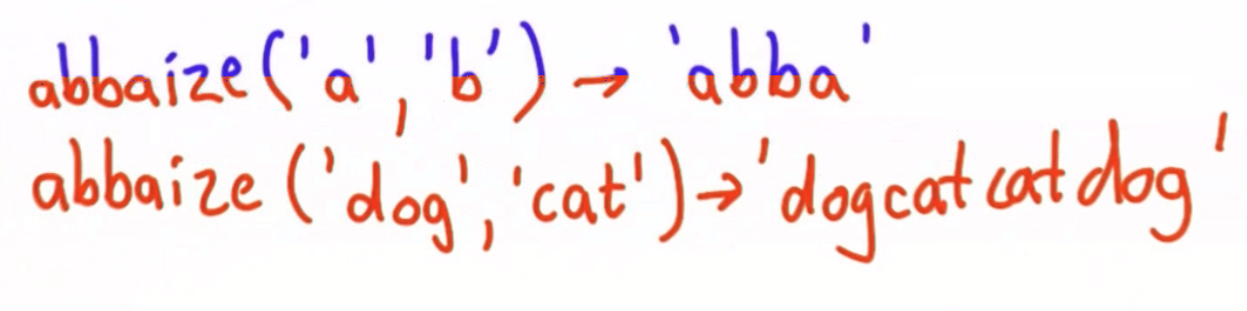
This is to distinguish it from the example, sum procedure, that we saw earlier that only takes 2 inputs. What sum3 should do is output the sum of those 3 inputs. So as an example, if you define sum3 correctly, if we print out the result of sum3, passing in 1, 2, and 3 as the inputs, the output should be the result of 1 + 2 + 3. Last I checked, that was 6.

Answer:

Here's the answer. We can define sum3. We're going to use the procedure definition syntax. So we have the def, followed by the name sum3, and now we need to find our parameters. Since there are 3 inputs, we need 3 names. We can make up the names whatever we want. I'm just going to use a, b, and c. Using letters like a, b, and c for inputs is usually not a good idea. They don't tell us a lot about what they're supposed to mean. In this case, since the inputs are just 3 numbers, and they don't mean anything special, we don't have any different meaning for the 3 numbers. Using a, b, and c is sort of sensible. But for most procedures, we want to use names that gives us a good idea of what the inputs are supposed to mean. Now we have our 3 inputs. We can produce the output right away just by adding those 3 numbers.

Programming Quiz: Abbaize

For this quiz, your goal is to define a procedure. We'll call it abbaize. This is very useful if you need to come up with names, and what it does is take 2 strings as its inputs and outputs a string that is the first input, followed by 2 repetitions of the second input, followed by the first input. As an example, if we called abbaize, passing in the string 'a' and the string 'b'-- single character strings-- what we should get as a result is the string 'abba'. If we called abbaize, passing in the string 'dog' and the string 'cat', what we should get is the string 'dogcatcatdog', and there aren't any spaces between the strings. They're just all pasted together.

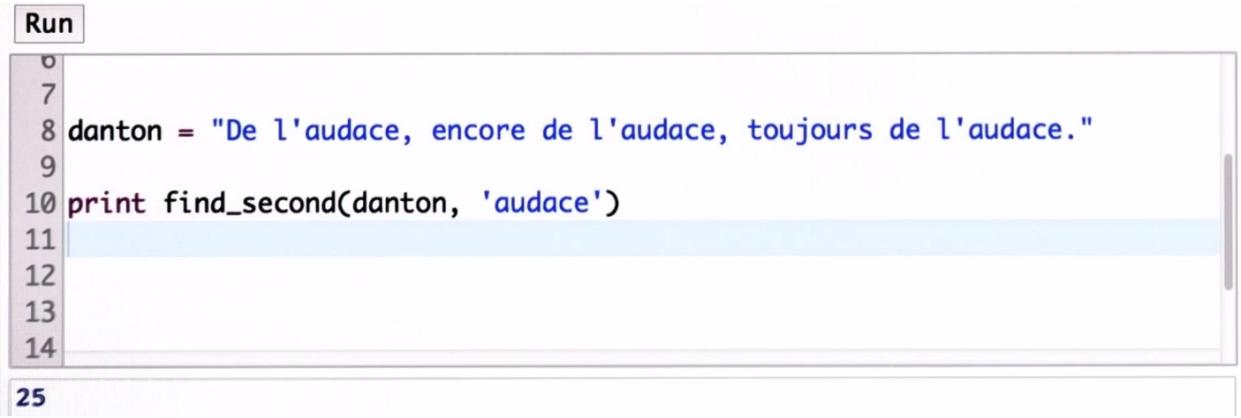


Answer:

Here's one way to define it. We'll call the inputs a and b, and all we need to do is produce a string concatenating them all. We can do that with one return statement. We're going to put a + b + b + a as the result. Now let's check that it works, passing in a and b. We run, and we get the result 'abba'. We can try the other example, passing in 'dog' and 'cat'. and now we get 'dogcatcatdog'. So far all of the procedures we've defined have been pretty simple. They've only required 1 line, maybe 2 lines of code. For the final quiz of this section, we're going to ask a much more challenging question.

Programming Quiz: Find Second

So far all of the procedures you've written have been quite simple, only needed a line or maybe two of code to finish them. Now we're going to try a quiz which is going to be much more challenging, and I'm going to give it a gold star to show that this is an especially challenging quiz. If you can get this one on your own, you're really understanding things very well. If you get stuck on this one, that's okay. There will be some hints to make it a little easier, and then hopefully you'll understand the answer after you see it, and you'll have a lot more chances especially on the homework to try some harder questions on your own.



So here's the question. Your goal is to define a procedure, and we'll call this one "find_second." It takes 2 strings as its inputs, so the first input is the search string. The second input is the target string. And your goal is to print out the position of the second occurrence of the target string in the search string. It should output a number that is the position of the second occurrence of the target string in the search string. I'll show you an example in the Python interpreter of what find_second should do, then you should try and solve this on your own. For an example, here's the quote we had in unit 1 from George Danton, and if we evaluate find_second, passing in danton as the first input and 'audace' as the second, what we should get is the position of the second occurrence of 'audace' in the input string danton. When we run this, we see that we get 25, which is that position.

Answer:

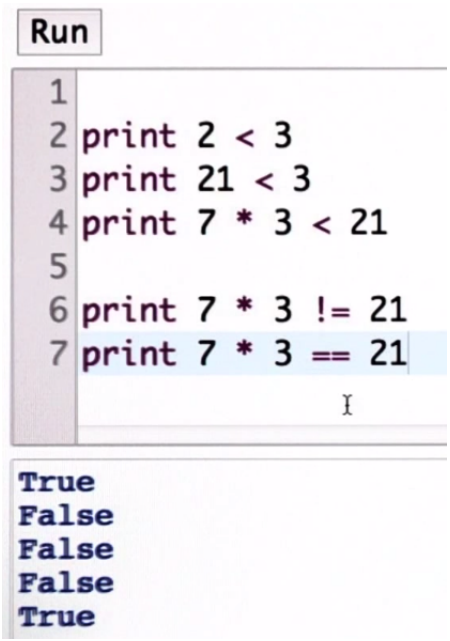
Here's a way to define find_second. So we're defining a procedure, giving it the name find_second. It has 2 inputs. One name for the input would be search and target. The names could be anything you want, but it makes sense to give them names that tell us what we're doing. And then the first thing we need to do is find the first occurrence. So we're going to use search.find, passing in target to find the first occurrence. And now we're going to store them in a variable. To find the position of the second occurrence we have to do search again, so we're going to call search.find. This time we need to have the second input, which is first + 1 to say find the first occurrence after this position. We'll store that in the variable second, and we'll return second, and so that was the implementation that was used in the test. We could make this a little simpler.

One way to make this simpler--we don't need to actually have the variable second. We could return second right away, and so now we only have 2 lines, and we'll run the same test just to show that it still works. We could even make it simpler than this. We don't even really need the variable first. We could replace first here with the search to find the first occurrence. Then we don't need the variable first. We only need 1 line. It's a good question, which of these is the best? I think probably the one before this was actually better than this. This is a complicated enough expression that it's hard to read. We're separating it into the variable first, and then using first here has exactly the same meaning, produces the same result. This is a little bit easier to understand what's going on, so I think I

prefer this version of the procedure. But either one certainly is okay, and either one works well.

Quiz: Equality Comparisons

Everything we've done so far has been pretty limited, that we had to do the same thing on all data. We couldn't do anything that really depended on what the data was. What we're going to do next is figure out a way to make code behave differently based on decisions. The first thing we're going to do is figure out some ways to make comparisons, so we have a way to test and decide what we should do. Python provides lots of different operators for doing comparisons. There are things similar to what we've used in math. We have a less than sign that compares two numbers. We have the greater than. We have a less than or equal to. Things like this... All of these operate on numbers, so we can have a number followed by a comparison operator, followed by another number. This is very similar to the grammar we saw earlier for arithmetic expressions, but now instead of having a plus or times here, we can have something that does a comparison. The output of a comparison though is not a number. It's a Boolean value, and a Boolean value is one of two things. It's either the value True or the value False. Let's see some examples in the Python interpreter. First, we'll use the less than to compare 2 and 3. So 2 is less than 3, so we expect the result to be true. When we run this, we see that the result is True. If we compare a number greater than 3, let's say 21 < 3. The result will be False. We can have any expression we want with a comparison, so we can do 7 * 3 < 21. When we run that, we also get False because 7 x 3 = 21, which is not less than 21. Another comparison operator we can use is not equal to. So != means not equal to. So 7 * 3 != 21 is False because 7 x 3 is equal to 21. If we want to do equality comparison, we don't use the equal sign, we use 2 equal signs. We call that the double equal. So now we have 7 * 3 == 21, and the result there is True. Now we're going to have a quiz to see if you can figure out why we need to use the == here instead of just the single =. The question is, why is the equality comparison done using ==, having 2 equals, instead of just a single = sign? The possible answers: Because = means approximately equal, and we want to do exact equality comparisons. Because we needed to use 2 characters for the not equal comparison, and we wanted the equal to be the same length. Because Guido, the designer of Python, really likes = signs. Because the single = sign means assignment, or it doesn't really matter. We can use either == or =.



[ ] Because = mean approximately equal
[ ] Because not equal uses two characters !=
[ ] Because Guido really likes =
[x] Because = means assignment

[ ] It doesn’t matter, we can use either == or =

Answer:

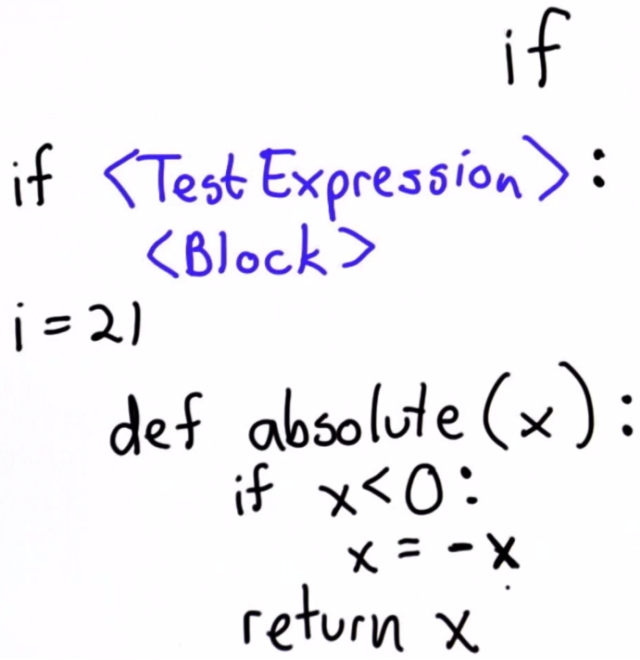
The answer is the 4th choice. We've already used the single = to mean something else. Single = means assignment. We can't use it to mean equality as well because that would be confusing. There are places where we couldn't tell what the code meant if single = meant both equality and assignment. Here's an example. The assignment i = 21 assigns the value of 21 to the variable i. The expression i == 21 is a comparison. It evaluates to either True or False depending on whether the value of i is 21.

Programming Quiz: If Statements

So now we know some ways to make comparisons. We want to use them to make decisions-- to make our code do something different, depending on the result of a comparison. The way to do that is to use an "if " statement. The structure of an "if" statement is: we have the keyword, "if", followed by a comparison, we'll call that the test expression, followed by a colon. And then, inside the "if", we have the block, and the block is the code that will run when the test expression is True. If the test expression doesn't evaluate to True, then the block doesn't execute. And, as was in the procedure definitions, we know the end of the "if" because of the indentation. All the statements inside the block are executed only when the test expression is True. The next statement that's not indented is going to be executed, whether or not the test expression is True. So here's an example of a procedure that uses "if": We can define a procedure, "absolute". It takes 1 input, which is a number. Inside the body of "absolute" we're going to use an "if" statement. We'll use an "if" test, where we're testing if the value of "x" is less than zero. So that's the test expression: we have the "if" followed by the test expression, "X < 0". In the Block, we're going to have one statement which changes the value of "x" to be "-x". The next statement, which will happen after the "if", whether the test was True or False, will return "x". So what this does is gives us the absolute value of the number that's passed in. If the number that's passed in is negative, we take its opposite and we use its negation, and then we return. If the number is positive, this test will be False so we don't execute the block that changes the value of "x" to its negation. We'll go right to the statement that returns "x". So now it's time for a quiz to see if you understand how to use "if" and can use it to define a procedure. Your goal

for this quiz is to define a procedure called "bigger" that takes in 2 numbers as its inputs and it outputs the greater of the 2 inputs. So here are a few examples: If the inputs are "2" and "7", the greater input is "7" and the output should be "7". If the inputs are "3" and "2", the greater input is "3" and so the output should be "3". If the two inputs are the same, say "3" and "3", the output should be "3".

Answer:



So there are lots of different ways we could define "bigger." Here's one possibility: We're going to make a procedure; we'll give it the name, "bigger". It takes 2 inputs. We can give them any name we want. Since there's no real meaning to the inputs, we'll just use "a" and "b". And now we want to use a test to figure out which one is bigger. We're going to use a comparison, so we're going to see if "a" is greater than "b". Then we want the result to be the value of "a"--that's the bigger one. So we'll return "a". If that's not the case--well, if it was the case, we returned "a"; if it's not the case, then we're going to get here, and we can return "b". So this way of defining "bigger" is a little bit asymmetrical. Given what you've seen so far, this is probably the best way to do it. There's another way to do that, that uses an extra construct that allows us to have 2 directions that we can use the "if" in. So instead of just having what we had before, where we have an "if" and we have a test expression, and we have a block that executes when the test expression is True, what we want to do is have something that executes when the test expression is False. And we can do that by using an "else". We can have--after the "if", instead of the next statement, we can have an "else" that's part of the "if". This means if the test expression was True, we're going to execute this block. If it's not True, we're going to execute the other block, which is inside the "else". As before, the indentation tells us when we're done with the block, so after the "else", whatever code is here will execute whether the test expression is True or False. The test expression determines whether we execute the block inside the "if" or the block inside the "else".

So let's see how to rewrite "bigger" using "else". We'll use "a" and "b" as the names of our parameters, as before, and we'll start the same--we're going to have our "if" and our test is still "a > b", and if that's the case, then we return "a". Instead of having the next statement return "b", we're going to think about having an "else" clause that says: well, if "a" was not greater than "b", what shall we do? If "a" is not greater than "b", then "b" is either the same size or bigger, and what we want to

do is return "b". So this means the same thing. In this case, neither one is particularly better. I think this one is a little more symmetrical. There are lots of cases where it makes things work a lot better by using "else". And I'll show you one way that this could be different. If we didn't want to have two returns-- maybe we'll want to do something else with the result, instead of just return it-- well, with the "if, else" we can do that. So we can have--inside the first "if", we'll assign to the variable "r" the result "a". That's going to be the one that's bigger. Inside the "else", we'll assign to the variable "r" the result "b". That's the one that's bigger when the "if" test fails. And now the next statement here could return "r". Usually, we like our programs to be shorter rather than bigger. In this case, when we're defining "bigger", maybe the "bigger" version is better because that's a little easier to understand and follow.

Programming Quiz: Is Friend

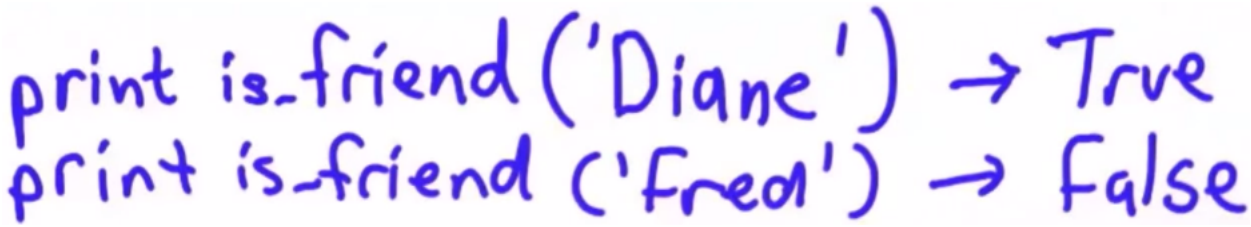
Your goal for this quiz is to define a procedure, is_friend, that takes a string as its input, and outputs a Boolean value indicating if the input string is the name of a friend. This would be very hard to do, in general, but we're going to assume that I have a very strict policy on friends. I'm friends with everyone whose name begins with the letter "D"--and no one else. So if the input name begins with the letter "D", your procedure should output True; otherwise, it should output False. So, for example, if we did: print is_friend('Diane'), the output should be True and if we did: print is_friend('Fred'), you should see the output False.

Answer:

So here's one way to solve the question: we're going to define a procedure, so we'll define a procedure, is_friend. It takes a single string as input. We'll call that the name--that's the name of the person. We're going to do a test using an "if" statement. We'll test if the first character of name is equal to an uppercase "D". If it is, that means the person's a friend, so we can return True. If it's not-- they're not a friend-- we don't need the "else", right--we could just have our procedure return False. I'm going to keep things more symmetric by using an "else" and if the name does not start with a "D", we can return False. So let's try that. With our example, ('Diane'), we get is_friend is True, and when we try with ('Fred'), we get the output, False. So this looks like what we want. This is actually a lot more code than we need. There's no reason we have to actually have the "if" expression. Since the test itself evaluates to a Boolean we could just return right away. We could return the result of the comparison. We're doing the equality test, if name[0]--the first character of name--is equal to "D". So this has the same meaning, much simpler thing. Sometimes people think it's unnatural to write expressions like this, that when you have test conditions, you feel like you need an "if" statement--but it's not required. It's much simpler, in this case, to just have the one line where we return the result of the equality test, directly.

Programming Quiz: More Friends

So for the next quiz, I'm going to make this a little more interesting and assume I'm not quite so picky about who my friends are. I'm going to be friends with people whose name starts with either "D" or the letter "N". So now the question is: we're still going to define the procedure, is_friend but I've changed my definition of a friend, so now we assume that I'm friends with everyone whose name starts with either the letter "D' or the letter "N", but no one else. So this result would still be correct--that would be correct. If we asked: is_friend a name that starts with the letter "N," let's say "Ned," then we should also get the result, True. So see if you can figure out how to define "is_friend" for this new definition.



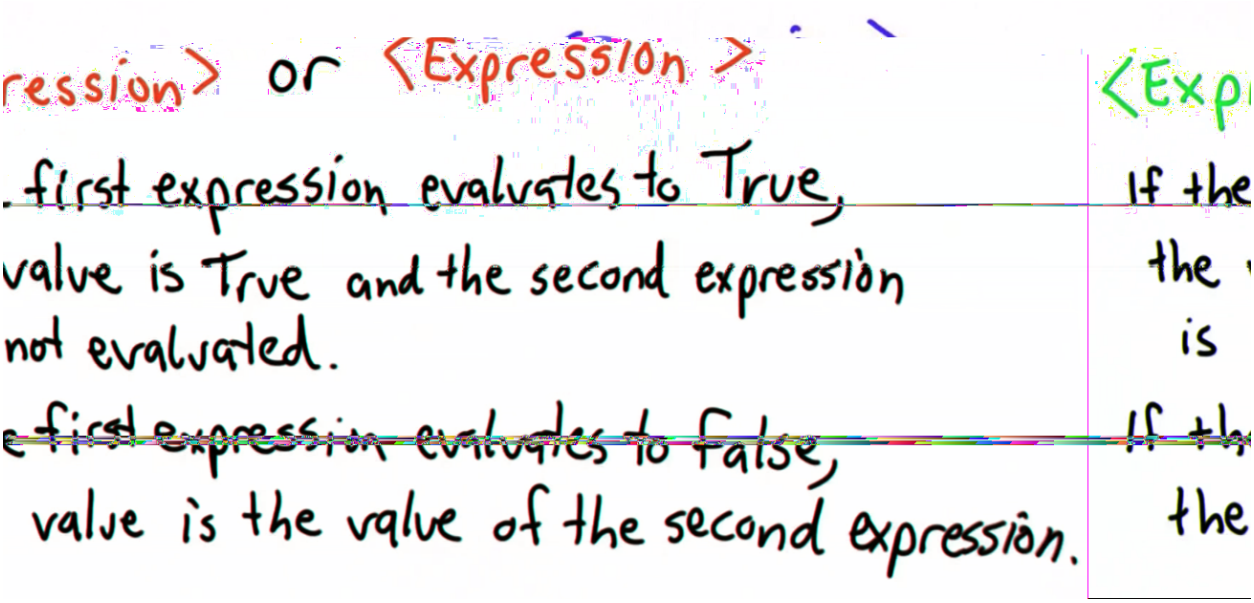
Answer:

So there are lots of different ways we could solve this. Here's one: so we're going to define the procedure, is_friend. First, we need to check if the name starts with a "D". If it does, then our result is True, and we can return True right away. If it's not, we're not done yet. We need to check if the name starts with the letter, "N". If it does, we return True. If we got to here, it didn't start with either a "D" or an "N", so that means we should return False. Let's see how that works. So we have 3 tests. The first one starts with a "D", so it should be True. The second one starts with an "N", so it should be True, and the third one starts with an "F", so it should be False. So there are lots of different ways we could have solved this. We could have used an "else" here--so we could have had an "else", and inside the "else" have the "if". And if it's not, then we could have an "else" here and the return, False. That will work, just the same-- a bit more code, but it also shows the structure perhaps a little better than the first one. The other thing we could do is return the result right away. To do that, we need to introduce a new Python operator that we haven't used yet, and that's the "or" operator. So we could do just this. This would be the equivalent of the long procedure we had before, where checking if the name at position zero is equal to "D" or the result of name at position zero is equal to "N". That produces the same result. And we've introduced the new construct, "or", which gives us the logical "or" of the 2 operands on its left and right.

Or

This will be the equivalent of the long procedure we had before. We are checking if the name at position zero is equal to D or the name at position zero is equal to N. That produces the same result, and we've introduced the new construct "or" which gives us the logic "or" of the two operands on its

left and right. So just to see how "or" works we can use "or" like this. So if we have "True" or "False", the value of "True" or "False" is "True." If we have "False" or "True", that also has the value "True." "True" or "True" is also "True", but "False" or "False" is "False". And I'm showing these boolean literals. This could be any expression we want, like we use the name at position zero here. The one important thing that "or" does is different from other operators: it only evaluates what it needs to.



So here's an important example. So we could say "True" or "this_is_an_error". That will also evaluate to "True" even though if I just look at "this_is_an_error" by itself that's an undefinable variable. We didn't define the variable "this_is_an_error" but when we evaluate the "or" expression because we already found that the first part was "True", we don't get an error. If we try to evaluate instead of "True" or "this_is_an_error" if we evaluate "False" or "this_is_an_error". It's necessary to look at the second operand and then we will get an error. So let's try that, we are doing "False" or "this_is_an_error" we get an error because since the first operand was "False" to know whether the "or" is "True" or "False", we need to look at the second operand and that's not defined. This behavior of "or" will turn out to be very useful. We'll see other places where we rely on the second operand not getting evaluated if the first one is "True." So to summarize, here's what the "or" construct looks like. So if the first expression evaluates to "True" then the value of the "or" construct is "True" and there's no need to evaluate the second expression and in fact the Python interpreter does not evaluate it. If the value of the first expression is "False" then the value of the "or" construct is the value of the second expression

Programming Quiz: Biggest

Okay, so we're going to have a last quiz for this section, and it's going to be a fairly challenging one. So we'll give it a gold star. Your goal here is to define a procedure, "biggest", that takes 3 inputs that are all numbers and it outputs the greatest of the 3 numbers. So here's a few examples: So if the 3

inputs are "6", "2", and "3" the output of "biggest" should be "6". If the 3 inputs are "6", "2", and "7", then the output of "biggest" should be "7". And if the 3 inputs are "6", "9", and "3", the output should be "9". So whichever of the 3 inputs is the greatest number, is the output that your procedure should produce, and it should return that. So I've shown the outputs. We wouldn't actually see them, unless we printed them out.

Answer:

So this is a pretty tough question--I hope you were able to solve it. There are lots of different ways to solve this. I'll start by working out one way on the sketchpad, and then we'll see a few different ways to solve it in the Python Interpreter. So I'm going to call the 3 inputs "a", "b", and "c". We can call them anything we want, and since they're not meaningful numbers, might as well just use the first 3 letters. So one way to solve it is to have a big, complicated, nested "if" statement.

So we're going to have several comparisons. First, we want to check if "a" is greater than "b". If "a" is greater than "b"--well "a" might be the biggest, but we don't know yet; we still have to check if "a" is greater than "c". So we need 2 "ifs" and the important thing to notice about the 2 "ifs"-- the comparison, "a greater than c" is indented inside the "if a is greater than b". So this will only happen when "a" is greater than "b". That means if "a" is greater than "b" and "a" is greater than "c", "a" is the biggest, so we can return "a". If it's not--well, now we've got the case where "a" was greater than "b". "a" is not greater than "c", so that means "c" is greater than or equal to "a". So now we know that "c" is the biggest since "c" is bigger or as big as "a" and "a" is bigger than "b".

So now we can return "c". So when this is not True, we know that "c" is greater than or equal to "a", and we know that "a" is greater than "b" because this test evaluated it to True. So we know it's correct to return "c". Now, we want an "else" that corresponds to this condition, so here if "a" is greater than "b", we evaluated this code. If "a" is not greater than "b", well that means we have: "b" is greater than or equal to "a". And in this case, we want to check whether "b" is greater than "c" so we need another "if"--I'm going to use "if b is greater than c". Now we know that "b" is greater than "c". "b" was greater than or equal to "a" because we went to the "else" here. So that means we can return "b". If not--well then, we know "b" is greater than or equal to "a", and "c" is greater than or equal to "b", which is greater than or equal to "a". So that means "c" is the biggest. We should return "c". So here we have a definition of "biggest". It's pretty complicated. We have 3 different "if" statements. The main one that starts by comparing "a" and "b", and then within that, we compare "a" and "c". Then we have the "else," and within the else, we compare "b" and "c." Whenever we have lots of nested "if" expressions-- and certainly, we can have more than this amount, but this is already getting to be enough to be confusing-- we should try to think if there's a clearer way to do this, a way to make the code smaller and simpler, easier to understand--and if it's easier to understand, that means it's also easier to write it correctly.

So the easier way to write this would be to use the "bigger" procedure that we defined earlier in this unit. And if we remember what that was, we defined "bigger" like this: "bigger" took 2 inputs, it compares them: if "a" is greater than "b", it returns "a". If not, well then "b" is bigger--or at least equally big-- so we can return "b". So if we have "bigger" defined, which we already did, then we can define "biggest" in a much simpler way. We don't need all these "ifs". All we need to do is use "bigger" twice. So if we remember that we have the "biggest" procedure, well then we can think about "biggest" this way. So we've got 3 inputs coming into "biggest", and it should produce 1 output--the biggest of those 3. Well, we've got the "bigger" procedure so we could use "bigger" to compare two. If we put "a" and "b" into "bigger" the output here is going to be the bigger of "a" and "b". To know the biggest of the 3--well now we need to compare that one with "c", so the inputs to this ""bigger" procedure will be the result of "bigger" of "a" and "b" and the input, "c". And the output of that will be the biggest of the three. So this is composing 2 calls to "bigger". We want the inputs to the first call to be "a" and "b". We want the output of that call to be one of the inputs to the next call. So here's how we can do that, in code. We can return the result directly, and the result will be the result of "bigger"-- and this is the last "bigger". When we do composition, we need to think about the function that we actually do last coming first because that one needs the inputs of the other ones. So the input to this call to "bigger", it has 2 inputs. It has the result of this one, which took "a" and "b" as its inputs. So that's what's here, taking in "a" and "b", producing this output. That's one of the inputs, and then the second input to "bigger" is "c".

So that's a much shorter way to write "biggest". It takes advantage of the fact that we already defined a procedure that does this for 2 inputs. And now we want to do the "biggest" for 3 inputs. There's actually an even easier way to do it, and that's using a built-in operator. There's a built-in operator, "max", and we could use "max" directly to implement "bigger", just returning the "max" of a, b, and c. If we actually knew about the built-in operator, "max"-- well then, we wouldn't need to find "biggest" at all. We could just use the built-in "max". But the important thing to see here is that we can define procedures ourselves. And, in fact, we've seen enough at this point that every built-in procedure in Python, you could actually define yourself. And it's even better than that, that you actually know enough at this point, that you could write every possible computer program, using just the things that we've seen. And this is a pretty astounding result, and I mean this in a very strong sense, that everything that could be computed mechanically by any machine can be described using a program that only used the things that we've seen so far.

All you need is procedures, simple arithmetic with the comparisons, and "if" statements. And this is a pretty amazing thing. This was shown by Alan Turing, back in the 1930s. Alan Turing is probably the most important computer scientist. In the 1930s, he developed an abstract model of the computer which we now know as the "Turing Machine". And he proved that that machine, with a very few simple operations, could simulate any other machine. I should point out that Alan Turing was doing this back in the 1930s, when there weren't computers like we think of today. In the 1930s, what someone thought a computer was was a human who did calculations, and they did calculations in

a mechanical way, following steps such as what you might have learned in grade school to do long division or long arithmetic. Alan Turing showed that a very simple model was enough to capture everything that a mechanical computer could do, whether it was a human computer or a modern computer that operates electronically, like we think of today. Following World War II, he worked at Bletchley Park.

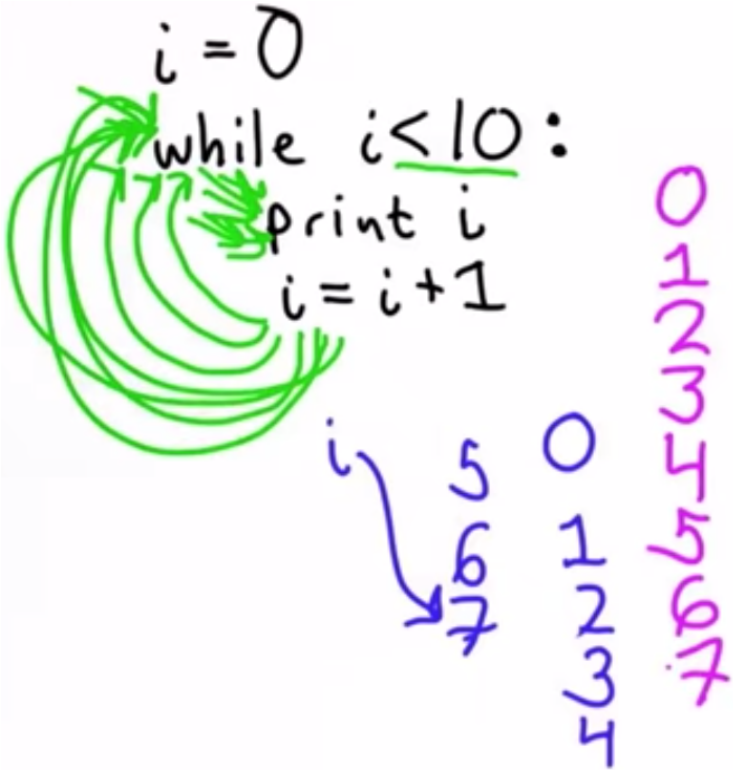
This was the headquarters where all the British efforts to break the Nazi encryptions were done. Alan Turing lead the effort there to break the Enigma Code. Enigma was the most widely used cypher code by the Nazis. And Alan Turing built machines that could be used to break the Enigma Code. These were sort of like computers. They would do lots and lots of calculations. The big difference between these machines and what we think of as computers is these machines weren't programmable. They were built for one very specific task, for doing a calculation that was useful for breaking Enigma encrypted messages. And it was tremendously useful that they could do that. This had a huge impact on WWII. But they weren't computers because they couldn't be programmed to do anything else. We're not going to get into the theory of Turing Machines in this course, but I hope you'll take one of the later courses where we do. The important point to make now is that, with a very few simple operations, you can simulate any other operation you want. And you've seen arithmetic, you've seen comparisons, you've seen how to define and call procedures, and you've seen how to use "if" to make decisions.

This is enough to simulate anything else a computer can do. So with just that, you could write every possible computer program. We could end the course here and you could, from just the things that you know now, build any computation you want. We're not going to end the course here though. And the reason for that is, although you could, in theory, build every computation using just these very simple things that we have so far, that's not going to be the best way to build computations, that there are more powerful constructs that we can use that mean, with a smaller amount of code, or in a more elegant way, that we can write the computations we want. We also have a lot to learn to get practice using these to define more interesting programs. But in theory, everything that you need to know to write any possible computation, you already know, from just these constructs that we've seen so far.

Quiz: While Loops

I said we didn't need any new constructs, but we are going to introduce one that will make writing the code we need for a search engine much more convenient. It's also very useful for lots and lots of procedures that we want to write, and the construct we are going to introduce is called a loop. It's a way to do things over and over again. The kind of loop that we are going to introduce first is called the while loop. The syntax for the while loop is this: we have the key word "while", followed by a test expression followed by a colon, and then inside we have a "block" and the "block" is a sequence of instructions. This is very similar to the "if" statement. So as a reminder, here's what the "if" statement

looks like we had an "if" followed by a test expression, followed by a colon and then an indented "block", which is a list of statements that executes whenever the test expression evaluates to a "True" value. With "if", the block executes either zero or one times, depending on whether the test expression is "True". With "while", the block can execute any number of times. It keeps going as long as the test expression is "True". So with the "if " statement, if the test expression is "True" we go to the block, run the block and then continue. If the test expression was "False", we go right to the next expression. With a "while", we also start by checking the test expression. If it's "True", we go to the block. But now, instead of going to the next statement, after the block, we go back. We try th back to the block. We always go back to the test expressio back to the test expression. If it's "True" again, we do the expression, and we can keep going around as many times is "True" we'll keep executing the block and keep trying t the test expression is "False." Once the test expression is



So this means a while loop can execute the block either zero times if the test expression was "False" at the beginning, one time if it was "True" the first time, but "False" after that, two times, three times...any number of times, it could keep going forever. There's no requirement that guarantees the test expression eventually becomes "False". So here's an example of while loop. We start by initializing variable i, and we'll give it a value, zero. Then we have the "while", the test expression says i is less than ten. So that means as long as this evaluates to "True", we'll evaluate the block. And what the block does is print i and then add one to i. So here's what happens when this executes. So initially, the value of i is zero, so i is less than ten. So that means we'll enter the loop. So that means the test expression is "True", so we'll enter the block, we'll print i. So we'll see the value zero printed and then we'll do the assignment, so that will change the value of i. We add one to i, so that's gonna be making the value of i now refer to one. So if it was an "if", we would be done now, but because it's a "while", we keep going. We go back, test again, if i is less than ten. Now the value of i is one, which also is less than ten so we continue, go to the block again we print i, this time we'll see the value one. Then we go to the next statement, increase the value of i by one. That makes the value of i two. Now i refers to the number two. Because it's a "while", we keep going, we go back to the test expression. i is less than ten, still less than ten, now it's two, we are going to print the two. We are going to add one. That will make the value of i three, and we are going to keep going. Test again, and i is still less than ten, so we are going to print i again, we will print the value

nine, then we add one, so that's going to have one to nine, we'll get ten. So that's the new value of i and we go back again and now we do i is less than ten, and now i has the value ten. Ten is not less than ten, so that will be "False" and we are done with the while loop. We'll continue with whatever statements here, in this case, there is none. So in the loop ``what we've done, we've gone through it ten times and we've printed the numbers from zero to nine.

The new value of i will be ten, if we do anything here that uses the value of i we'll see that the value of i is ten. So to see that you understand while loops, we'll have a quiz. So the question is, what does this program do. Here's the program. We start by assigning zero to i. We have a while loop, where the test is not equal to ten. So the test is i is not equal to ten. Then we have i equals i plus one, so we are assigning to i the value of i plus one and then we are printing i. So this is similar to the example, but different in a couple of ways. So it's up to you to see if you can figure out what the program does. Try to figure out yourself. You can certainly also try running this in the Python interpreter. The choices are produce an error, print out the numbers from zero to nine, print out the numbers from one to nine, print out the numbers from one to ten, or the final choice is it runs forever or at least until our machine runs out of power. So see if you can figure out what it does. You can definitely try running it, but try to figure out on your own before running it in the Python interpreter.

[ ] Produce an error
[ ] Print out the numbers from 0 to 9
[ ] Print out the numbers from 1 to 9
[x] Print out the numbers from 1 to 10
[ ] Run forever

Answer:

So the answer is: it prints out the numbers from 1 to 10. And if we follow through the code, we can see why. So initially, the value of "i" is zero. The test says, well "i" is not equal to 10, so zero is not equal to 10-- so we go through the Block, and the Block adds 1 to "i". So that will change the value of "i". Now "i" refers to "1". And then the next statement prints the value of "i". The value of "i" now is 1, so it will print, "1". And then, because it's a while, we go back. We do the test again--I'm not going to step through every time, so we keep doing this. As "i" gets bigger, we're going to keep going. Eventually--let's say "i" is 9, and by the time "i" is 9--well, after "i" was 9--we print "9". So we've printed the numbers from 1 through 9. At this point, "i" is still not equal to 10, so we still go through the loop body. Now we add 1 to "i"--that will make the value of "i" 10. Then we do the print, so that will print out, "10". Then we go back to the "while" test. Now the value of "i" is 10, so: "i" not equal to 10--"10" not equal to 10-- is False, because 10 does equal 10. That means we're done with the "while" loop and we're going to continue, but there's nothing to continue with, so we're done. So the net result of the code is printing the numbers from 1 to 10 and that's all that happens. So now the test is False. We don't execute the Block anymore, we would continue with the next statement. There isn't one, so we're done and what the code did was print out the numbers from 1 to 10.

Quiz: While Loops 2

So we're going to have another quiz about "while" loops. This one's going to be a little trickier. The same idea, your goal is to guess what this program will do and, like the last one, try to figure out the answer yourself rather than by running the code. We're going to assign 1 to the variable "i". We'll have a "while" loop where the test condition is: "i" is not equal to 10. And, in the Block, we'll have a statement that says: "i" is assigned the value of "i" plus 2-- so adding 2 to "i", then we'll print "i". So the choices are: produce an error, print out the numbers: 2, 4, 6, 8; print out: 1, 3, 5, 7, and 9; print out: 3, 5, 7, and 9--or run forever.

[ ] Produce an error
[ ] Print out 2, 4, 6, 8
[ ] Print out 1, 3, 5, 7, 9
[ ] Print out 3, 5, 7, 9
[x] Run forever

Answer:

So the answer is: this loop will essentially run forever. It won't run till the end of the universe, because we'll probably have something else happen to our computer before that. But there's no stopping condition for the "while" loop. The "while" test will never become False. We start with "i" equals 1. We keep increasing by 2, so the values that i will have will be 1, 3, 5, 7, 9, 11, and so forth. The loop test checks if "i" is equal to 10. If "i" is not equal to 10, we keep going. So if "i" is not equal to 10, we're going to keep going through the loop. "i" will never become 10--we skip 10. The loop test is never False--that means the "while" loop just keeps going forever and ever, never finishing. I'll show you what happens when we run that. And I'm going to show this running in the Python shell instead of running through the Web browser ID that you've been using so far in the class. And the reason for that is we can see more output.

We can run longer things-- the way you run the programs for class, there's a limit to how long they can run and how big the output can be. With the shell, there's no limit like that, so I can show you a longer execution. There are instructions on the Web site, if you want to run programs yourself in the Python shell. There are lots of different ways to do that, and instructions for how to do that on different platforms. It is not necessary to continue with the class, but it will make it easier to see what's going on here. So here's the code. I'm going to enter the code that we had, so we have "i" is assigned 1, and then the loop, "while i is not equal to 10". We're going to increase "i" by 2, and then we're going to print "i". So now when I type return one more time, this "while" loop will start running, and we'll see it print out the numbers and it will keep going as long as we let it keep going. So here we go. We see 3, 5, 7, notes keeping going, really fast. So at the top, you can see the first 5 numbers: it started at 3, 5, 7,

9, 11. Since it skipped 10, the loop didn't stop. This condition was never False. And it keeps going--and it will keep going for as long as we let it print or until our computer runs out of memory or power. So a loop that never stops like this is what we call an infinite loop. That means it keeps going forever, and if you run programs on computers, you've probably encountered them a lot-- not just in the program that you just wrote, that we just used for this quiz, but oftentimes, when your software hangs and gets stuck doing something--that's what's going on, that there's some mistake that a programmer made. We call that a bug--there's a bug in the program, and it's stuck in an infinite loop; it's never going to make any progress. It's got stuck in something where the test condition will never be False and it's going to keep going forever.

Programming Quiz: Print Numbers

So now we're ready for a quiz where you're going to write your own "while" loop, and you're going to do that in a procedure. So your goal is to define a Python procedure-- and we'll call it "print_numbers", with an underscore. And it takes, as input, a positive number, and it prints out all the numbers from 1 to the input number. So, for example, if we ran "print_numbers", passing in 3, what should happen is it should print out the numbers: 1, 2, and 3. So this one's going to require some thought, but you know enough to be able to do it. You're going to have to define a procedure, you're going to have to use a "while" loop, and you'll probably need to introduce a new variable. So see if you can figure out how to define "print_numbers".

Answer:

Here's one way to define "print_numbers": so we're going to make a procedure using "def". It takes 1 input. We'll use the name "n" for our input. That's the number that we go up to. Now, to do the loop, we need to have a new variable that keeps track of the loop iterations. We'll use "i" for that, and we start with 1--that's the first number to print out. We need the loop now, so we have the "while" and the test that we want is we want to keep going, up to and including the number, "n". So we want to print the numbers, all the way from 1 up to and including "n". So our test will now be a less than or equal to test. We want to keep going as long as "i" is less than or equal to "n". Once "i" gets bigger than "n", that's when we should stop. What we do in the loop body is print the value of "i" and add 1 to "i". We don't need to return anything; the only point of the procedure, as we were asked to write it, was to print out the numbers. So now let's do a test: we'll print the numbers up to 3, and there we get the output, printing the numbers 1, 2, 3.

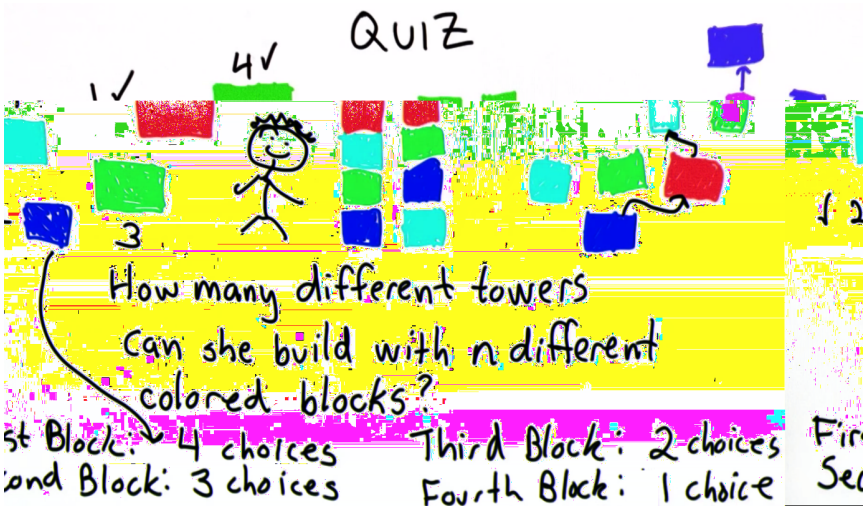
There are lots of other ways we could have written this. Another approach would be to start with "i" equals zero, make the test condition "i" is less than "n". So now we're stopping as soon as "i" is equal to "n". For this to print out the correct numbers, though, now we need to move the print. We want the print to be after we added 1 to "i", so the first number we print is supposed to be a 1, which will be the

case here. And the final number we print should be "n", which is the case here. Once the value of "i" is "n", this test will be False and we don't go through the loop anymore--so when we run this, we get the same result, printing 1, 2, 3. So as another test, let's see what happens when we pass in zero. And when we run this, it doesn't print anything. That's because when "i" is zero, "i" less than "n" evaluates to False. Zero is not less than zero, so we don't go through the loop at all and don't print anything--which seems like a pretty sensible thing. Since we said print the numbers between 1 and the target number, if the target number is zero, not printing anything sort of makes sense.

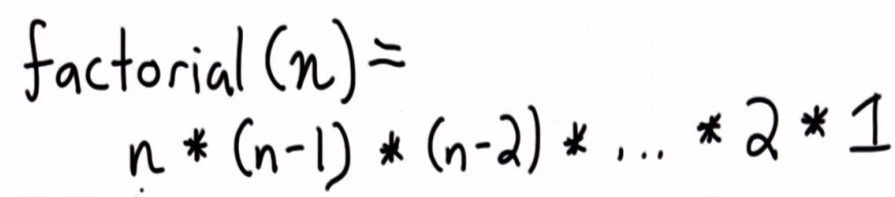
Programming Quiz: Factorial

For the next quiz, you are going to define a procedure that can help solve the problem of figuring out how many different ways there are to order objects. Here is the problem we want to solve. Suppose we have four blocks. We have a red block. We have a blue block. We have a purple block and we have a green block. We have a baby who wants to play with the blocks, and we want to know how long can the baby play with the blocks without getting bored. What the baby wants to do is try all the different ways of arranging the blocks. The baby will stack the blocks in a tower. The baby might pick the red block first ”put that on the ground. Then put the blue block on top. Then put the the purple block on top of that, and then put the green block on top. When she's made one tower, she is going to knock it down and make another one. She's going to keep doing this as long as she can make different color combinations. If she has to make the same tower twice, ”well, that's pretty boring. Maybe the next one she'll pick the purple block first, then she'll pick the red block, then she'll pick the blue block, and then she'll pick the green block. These two are different. The question we want to answer is, how many different towers can she build? Given that she has four blocks, we want to know how many different towers could you build with four blocks, but maybe she gets some more blocks. What we really want to know is ”given that she has "n" blocks, ”whatever number "n" is, ”how many different towers could you build where the color combinations are all different? The way to solve that is to think about combinatorics. The way to pick the first block, ”well, it could be any one of these four. We have four choices. Let's say ”for our example ”she chooses the red block first. Now it's time to choose the second block. Well, she can't use the red block again. She has already used it, so she can either use the purple, the blue, or the green block. She has three choices now. She can use the

purple block, the blue block, or the green block. Let's say she chooses the green block, so she is going to have the red block and then the green block. She had three choices. She could have chosen any of the other two, but now that she has chosen the red and the green, she only has two choices left for the third block. She can either use the purple one or she could use the blue one ”those are the only two that are left. Let's say she uses the purple one. Now for the final block well, she's used the red one, she's used the green one, and she's used the purple one. The only one that's left is the blue one, so she has no choice but to use the blue block. That means for the fourth block, there was only one choice. So, for each of these choices, she could choose any of the four for the first one, any of the three remaining ones for the second one ”regardless of what she picked for the first one. If she picked red, she could pick either purple, blue, or green, but if she'd picked ”say ”blue for the first one, then what she'd have left is purple, red, and green. She always has three choices for the second one. After that, there's only two choices left ”there's only two blocks left and she can choose either one. The pattern here is every time she chooses a block there's one less because she's used the block in the previous choice. For the total number of choices, what we want to do is multiply all these numbers.



To figure out the total number of choices, what we need to do is multiply the number of choices for the first block, and then we multiply that by the number of choices for the second and we keep going for however many choices there are ”in this case there are only four until we get down to one. Each time we make a choice, we have one fewer items left to use for the next choice. So the function that we are computing here is mathematically called factorial, and it's sometimes written with an exclamation point. A way to define factorial is ”for any value n, we can compute the factorial, which is the number of ways of arranging "n" items. We can compute that by multiplying "n" ”which is the number of choices for the first item ”times "n" minus one, ”that's the number of choices we have left for the second item, ”times "n" minus two, ”that's the number of choices for the third item, and so forth, all the way until we get down to one. So, your goal for this quiz is to define a procedure named "factorial" that takes one number as its



input ’’that's the number "n" here ’’and outputs the factorial of that number, which we can compute by multiplying "n" times "n" minus one all the way down to one.

Answer:

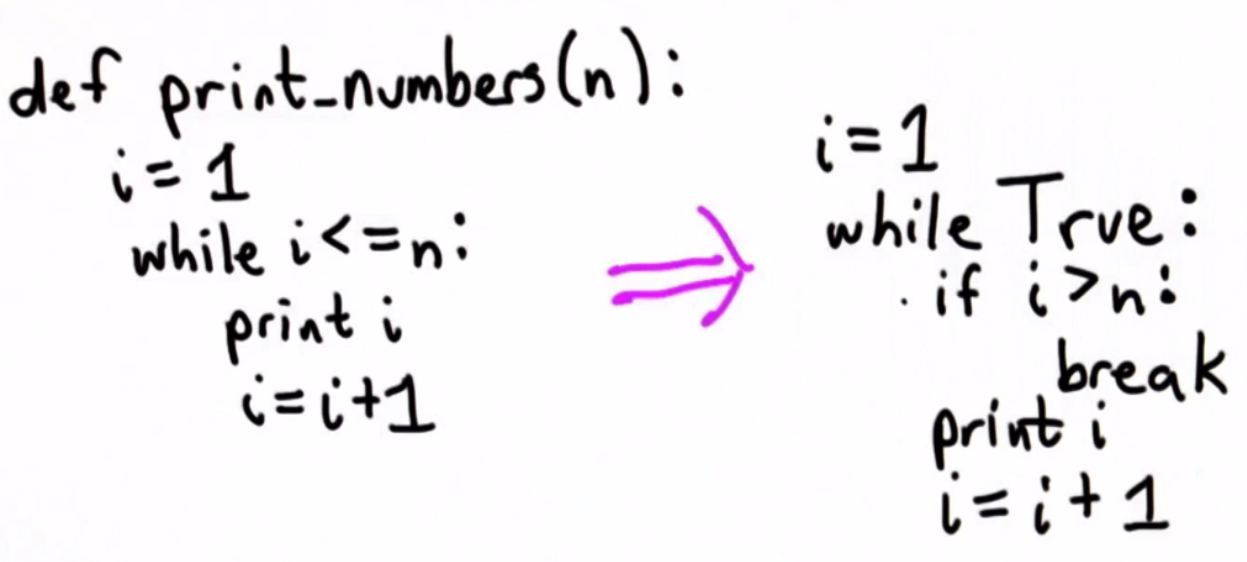
Here's one way to define factorial. There are lots and lots of ways to solve this, so we're going to define a procedure and its input is the number "n." What we want to do is compute this function ’’we're going to compute "n" times "n" minus one times "n" minus two. We are going to need a variable to keep track of the results as we go, as well as a variable to keep track of how many times we go. We're going to define a procedure that computes the factorial of "n." What we need to do is multiply all these numbers, so we're going to have a variable that keeps track of the results so far. We'll start by initializing that to one, so we'll call that "result" and we'll set its initial value to one. We're going to keep multiplying by "n" until we get to the end. We're going to have our "while" loop where the test condition is as long as "n" is greater than equal to one. We're going to start with "n" as the value that was passed in. We're going to subtract one each time we go through the loop until we get down to one, and keep updating the result. The result keeps updating, so we want the result to be the previous result times "n." We also have to change the value of "n" to keep reducing "n" as we go. We subtract one from "n" every time through the loop. The result is multiplied by the current value of "n". "n" is decreased by one. We keep going until we get to "n" is equal to one. We're not quite done yet. We want to return a "result," and the value that we want is stored in this variable "result," so we "return result" here. Let's try that in the Python interpreter.

Now we'll see what happens when we have the input 4, our child playing with the blocks. We see the result as 24. They're 24 different ways to arrange the four blocks. Factorials increase pretty quickly, so we could also use the factorial procedure to figure out the number of different ways to arrange a deck of playing cards. If we have a deck with 52 cards, there are 52 ways to pick the first card. When we run this, we see that the factorial of 52 is this really big number. That's why you can have so many different card games. Every time you play, the arrangement of the cards is going to be different.

Programming Quiz: Break

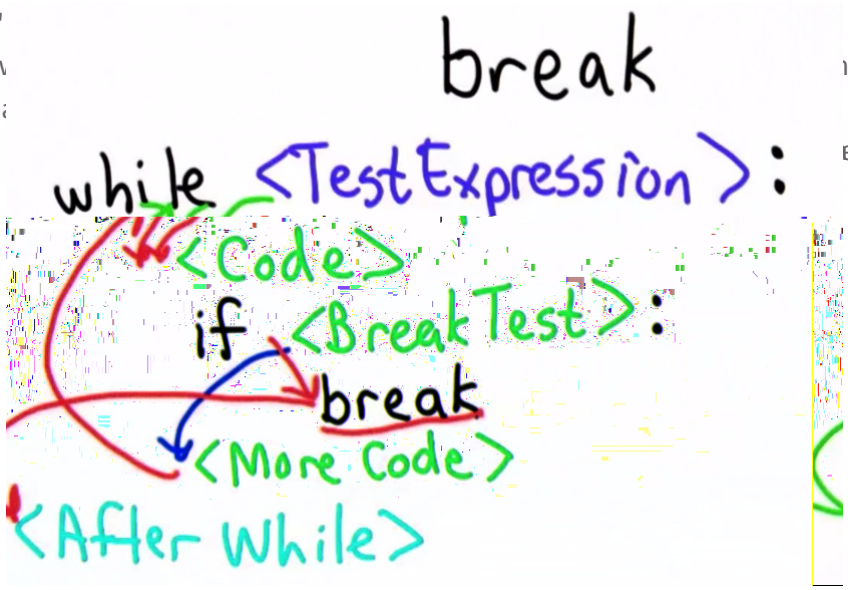
So now you have finished the hard quiz about factorial. You might think you deserve a break and you almost do, but we have one more thing to learn about "while" loops, and that's the "break" statement. "Break" gives us a way to stop the loop even while the test condition is true. Here's the typical structure of a loop with
a "break," so we have the "while"as we did before and we have the test expression and we had a colon. Our general "while" structure is we had a block here. Now we want to look inside this block a little bit to see the kinds of things that could be here. This is just an example of what might be in the block ’’we have some code, we have some Python statements, and then we have an "if" statement. That's going

to have another test expression we'll call that the "break test." That's a text expression that's checking whether it's tim
we have is "break" which ”by itself is all we need. What "break" means is stop executing the "while"
loop ”jump out of the "while"
loop and continue with the code after that. Whatever we had here, this is more code that was in
the "while" block. That will get skipped and we'll jump to this point which is the code after the while. So
what happens when we execute a loop with a "break," it executes like a normal loop. If the test
expression is true, we go to the code. If the "break test" is false, we don't execute the break. We would
continue with the "more" code. We would go back to the loop test and check the test expression
again. If it's true, we go run this "code." If the "break" is true, then we execute the "break."
What "break" does is jump out of the loop, so we don't execute the "more" code, we don't execute the test
expression again. If the "break" happens, what happens is we jump to the code after the "while." This
gives us a way to break out of the middle of the loop. Here's an example of how we might use that. Before,
we defined the procedure "print_numbers" without using break. We defined it like this. This is the
code we had before. We could rewrite that using "break." Now, instead of having the "while test" stop
the loop, we'll make the "while test" True. True will never become False. ”That means, if we didn't
have a break, the loop would keep going forever, but we're going to add a "break." We're going to
have an "if test" that says the stopping condition now is we're going to stop once "i" is greater
than "n." The way to stop is to use "break." If "i" is not greater than "n," then we're going to keep going. We'll do the sa
to "i". This loop has exactly the same meaning as the one we had before.



The previous one's better in every way. This is more complex. There's more code. This illustrates how
we can use break. We wouldn't want to do it this way, this is much worse than what we had before. We
want to keep our code simple and easy to understand. We'll see once we get to the code for extracting
all the links on the page that there are cases where it's easier to write the code using "break" than it is
without that. Now we are going to have a quiz to see that you understand "while" loops, as well
as "break." This is going to be a pretty tough quiz that you have to think carefully about how "while"
loops are defined and what "break" means. The question is which of the following are always

equivalent to "while," any test expression "T", and a block ”which is any statement "S." "T" and "S" could be anything, and your question is to understand which of these are equivalent to that. Here are the four choices. The first one is a "while" loop with a nested "if." a "break." The third choice is a "w a nested "if", an "else," and a "brea t" eent, "S"



Answer:

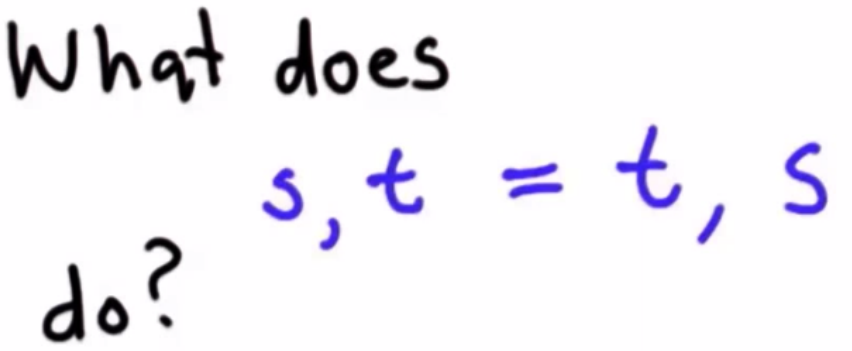
Here's the answer. The first one is equivalent. The reason for that is we have our code here that has "if False." "False" will never be true, so we'll never actually run the "break"” we'll never reach that code. We'll just run the statement again and we'll have the same thing that we started with. Every time the test is True, we execute the statement. The second one is not equivalent, and the reason for that is the "break" will stop the loop. If the loop would execute more than once in this version, and if we go through the loop twice, in this case we can only ever go through the statement once, since after we go through the statement once, we reach the "break" and then we're done. The third one is not equivalent, so here we're having a "while True" that will run forever except for we have the "break" and that would stop it. The problem is the test condition for the break is the same as the test condition for the "while," but it should really be the opposite. In the "while" we keep going as long as the test condition is True.

The way I have written in the code here, if the test condition is True, we stop. For this to be equivalent, what we should have written instead would've taken the opposite of the test condition, so if we had "if" not",” which changes True to False and False to True, then the test condition, then the break ”this would be equivalent to the original loop. What we have is not. The final one is the most complicated. This one is actually always equivalent to the original loop. The reason for that is if we think through the execution, so in the original loop, if the test condition is true, we ask to give the statement "S," and then we keep going. If the test condition is true the second time, we execute "S" again and we keep going. If we look at what happens here ”well, if the test condition is True, we execute "S." Then if the test condition is True again, we execute S and we keep going. That has the same behavior. We should trace it through the case where the test condition is not True. Here ”if the test condition was True, we execute "S." If it's not True the second time, then we're done with the loop and we continue. In this case if the test condition is True the first time, we'll execute "S." If it's

not True the second time, then we would go to the "else." So if it's not True, we go to the "else" and then we go to the "break" and the "break" gets out of the loop. We have the same exact behavior where we executed "T" once, then we executed "S," and then we executed "T." We didn't execute "S" again, and we broke out of the loop to continue. So these two are equivalent and the other two are different.

Multiple Assignment

So, break time is over; now, you know about "Break;" you know about "while" loops; you know about "if" to make decisions; you know everything you need to finish the goal of today's unit, which is to write the code for extracting all of the URLs from a web page. First, let's remember what we were doing before: we had the code that kept repeating to get one link after another. So, we had code like this: we had finding the "start_link"-- and we assumed, before entering this code, that "page" has the contents of some web page. We would search for the start of a link; then we would find the "start_quote," again using "find," but this time passing in the start of the tag and searching for the double quote. Then, we would find the "end_quote." And then we would extract the URL using the string selection-- selecting from "page," from the "start_quote" to the "end_quote." Then, we would print out the URL, and this was what we needed to do for one URL. We would continue by advancing the "page" to go from where we found the "end_quote" to the end; and then we would do all the same thing again to find the next URL. And we'd keep doing this code, over and over again, finding as many URLs as there were on the page. So, here's the code, shrunken down so we have some more space. What we want to do is extract into a procedure, this part [indicating the first four lines after "page" is defined]; and then we want to make it so we can keep going. We've seen "while" loops give us a way to keep going; so, we'll end up writing a "while" loop that keeps looking for links until we reach the end of the page.



So, the first thing we're going to talk about is how we extracted this procedure, and we're going to need to make a few changes to it, to make it work well in our "while" loop. We defined a procedure, "get_next_target"-- that takes the page as an input-- and essentially had exactly the code we have here-- except for, at the end, it would return both the URL and the value of "end_quote." And we had a quiz earlier where you determined that what "get_next_target" should do is return both the URL and the "end_quote"-- that is the position where the end of the quote was found. So, we needed

this procedure to return two values because we need to know both the URL, as a string, and where to continue looking for the next target. We haven't yet used a procedure that returns two things, but it's actually quite convenient and simple to do this. We can do this by just having two values on the left side of an assignment statement This works in general; so we can do-- instead of our simple assignment rule, where we had a name, followed by an equal, followed by an expression, and that would assign the value of the expression to "name"-- we can have multiple names on the left side, and multiple expressions on the right side.

So, we can do multiple assignment, where we have any number of names, separated by commas, an equal sign, and then any number of expressions, also separated by commas. As long as the number of names and the number of expressions match, what happens is the value of the first expression is assigned to the first name-- so this name [indicating] will now refer to whatever this value was-- [indicating] the value of the second expression is assigned to the second name. As an example, if we had "a, b = 1, 2", this would assign to "a" the value "1," and to "b" the value "2." In order to get the two values returned by this procedure, what we need is to have two variables on the left side and the procedure call on the right. The syntax we'll use is just the URL [url]-- we'll use that as the variable to hold the URL; and "endpos"-- we'll use that as the variable to hold the end position; and, then, on the right side, we have "get_next_target" passing in "page." Since this returns two values, it's like listing two expressions: the first value returned will be assigned to "url," the second value returned will be assigned to "endpos." Being able to have multiple variables on the left side of an assignment is very useful. Let's have a quiz to see an example of why.

Quiz: Multiple Assignment

This quiz is going to require a little bit of guess work, for how a multiple assignment works. I think you'll be able to bet the right answer. You can also try experiments in the Python interpreter. The question is for any variable--so let's assume s and t are names, and they've been assigned some value, and now we execute the statement s, t assign t, s. What does that do? The choices are nothing--after the assignment s and t both have the same value as they did before. That it makes both s and t refer to what the original value of t was before this statement. That it swaps the values of s and t so after the statement, s now has the previous value of t and t has the previous value of s. Or it's an error.

[ ] Nothing
[ ] Makes s and t both refer to the original value of t
[x] Swaps the values of s and t
[ ] Error

Answer:

So, the answer is, it actually swaps the value of "s" and "t." This is quite convenient, and the important reason for that is both the values on the right side get evaluated first-- to get turned into their values-- before we do any of the assignments. So, what's going on: "s" refers to some value-- let's call that "alpha"-- and "t" refers to some value-- we'll call that "beta." This is before the assignments. To evaluate the multiple assignment expression, we find the value of "t" here [indicating the second "t" in the expression]-- that has the value of "beta"-- and we find the value of "s"-- and "s" has the value of "alpha." We've evaluated both of these, and we've turned them into the values the names refer to; and, now we do the assignments. So, "s" will be changed to refer to the first value-- which is "beta"‘ and "t" will be changed to refer to the second value. This has very different behavior than if we did two assignments like this; so, if we tried, instead, doing "assign t to s," and then "assign s to t," that wouldn't swap the values, because, by the time we do the second assignment, the value of "s" has changed-- now it refers to whatever the value of "t" was. If we had done these two statements {indicating "s = t" and "t = s"], both of them would end up having the value "beta."

Programming Quiz: No Links

Now we have our get_next_target procedure, and we know how to use it by using multiple assignment to get the two results. There's one serious problem with the get_next_target procedure that we have to fix before we go on to the problem of outputting all of the links in the page. The problem is we didn't really think very carefully about what should happen if the input does not have another link. Let's try and see what actually happens the way we've defined it. Here's the code we have for get_next_target, and we're going to try an example. First we'll try an example where there is a link, so what we should get as the next target when this is the page that's passed in is the link that's included, which is between the double quotes inside the href tag. When we run that, we get our 2 outputs. We got the string that's the link, and we got 37, which is the position of the end quote. And we can see it's printing out the 2 outputs as a tuple, meaning the first one, the second one surrounded by parentheses, and we could get those in an assignment like this. Instead of just printing it, we'll use our double assignment, getting those outputs, and we'll print the value of just the URL. And we get the string http://udacity.com. That worked well.

Now what happens if instead of passing in a page that actually has a link, we pass in something that doesn't. Let's say our page is just the text "good." Now, there's no link to find, but the code is going to run anyway, and what we see as the result is we actually got the result "goo." Probably not what we wanted, and the reason we got the result "goo" is you remember the find command, if it does not find what it's looking for, it returns -1. And when we use -1 as an index, that means the last character of the string. So what we end up with is getting all the characters except for the last character. If we pass in something else that does have a double quote in it, so now we'll pass in 'Not "good" at all!' we see what we get back for the URL is "Not". This is not very useful. This is pretty confusing behavior. It's going to be very hard to tell when we've gotten to the last target because maybe "Not" could be a valid URL, and we don't know that. What we want to do is make get_next_target return something

more useful in the case where the input does not contain any link, and the way we're going to do that we'll leave partly up to you.

But I'll give you a hint to get started. We're going to want to change the code, and you're going to want to put something into the code here. If we found a link, well, we've got a value for start_link that's the position of the start link. If we didn't find a link, what we got back from find was -1, and I want you to change what the get_next_target procedure does, so when there's no link to find, what we should return is the value None. This is the special value that means there isn't anything. And we'll return 0 as the endpos. Let me write that out more precisely, but the hint is you're going to have to change the code after you've found the start_link to figure out whether there is a link to find. Your goal for this quiz is to modify the get_next_target procedure that we've defined so that if there is a link, it behaves exactly as it does now, still returns the URL of the link target, as well as the number of that position where the end quote is. But if there's no link in the input string, it should output None as the first output and 0 as the second output, and None is a special value that means it's not actually returning anything.

Answer:

So here's the solution. So we want to add an "if" statement because we want to make a decision. We want a test If the start_link is equal to -1. So that means that we did not find what we were looking for find returns to -1, so the value of start link is -1. Well, then we don't want to do the rest, we don't have an URL to look for. What we want to do is return the special value "None" and zero. So that's the change. Now when we run this with this string that does not have a link what we get as the output for the url is "None". We can't tell that's not a string, the way it prints out, but it's different from a string and one thing that we can do with "None" is sort of like "False". So if the value of the test expression in an "if" is "None" that's treated the same way as "False". So if url returned here is "None", then we'll print "Not here!". I guess that means we are here, but we are not "Here!". So here's what we get, we see that we got "Not here!" when we passed in a string that did not contain a link tag. If we go back to the string that has a link tag and now we run the code, we'll see that the url was some string that's treated as "True" and we get the print "Here!". So the test condition for a boolean doesn't necessarily have to be just "True" or "False". Python will interpret different values as "True" and "False" and the important thing here is that any string value other than empty string is treated as "True" and we'll go through this part of the "if". The value "None" is treated as "False" and we will go to the else.

Programming Quiz: Print All Links

So now we've modified our get_next_target procedure, and it will return None as the first output if there's no next target. Otherwise it will return the URL and the end_quote. And now all we have to do is figure out how to keep going. Let's look at what we were doing before. We turned these 4 lines into

our procedure so now we can--instead of having all this code, we're going to call the get_next_target and assign the results. Now we've got the value of URL, which we do on a print out, so we're still going to print that out as before. And we still want to update the value of page, but we don't want to use end_quote now. What we want to use is the value that was returned here, which we assigned to the variable endpos. And then, well, we're doing it all again.

So, all of the code here is just another call to get_next_target, and similarly, this would be endpos instead of end_quote. And we want to keep going until we get to the end, so how do we decide when we've gotten to the end? Well, we got to the end when the URL that's returned is None. And when the call to get_next_target returns None for the URL, we know we're done. We've seen a way to keep going. That's a while loop. We've seen a way to do a test. We want to test the URL. And we have everything we need now to print all the links on the page. I'm going to give you a start for how to write this procedure, and then you're going to finish it. What we're going to do is define a procedure that will print all the links on the page, and it takes the page as input, and so we want to use a while loop to keep going, and I am going to leave blank the test condition for the while loop. We'll leave that as something for you to figure out. In the body of the while loop, what we're going to call get_next_target, assigning URL and endpos to the result.

This is just like what we were doing in the example code. Now what we need to do is check whether we got a valid URL, so what we were doing here, we assumed that we always got a valid URL, and we printed it out, and we kept going, but we need to do something to test whether the URL that we got back is None. That's what we'll do. We have if URL, and if that's True, that means we found a valid URL. We didn't get the value None as the result from get_next_target. And so we want to print that out as we were doing before. We'll print the URL, and we'll advance the page to the next position. If we didn't get a valid URL, that means that get_next_target did not find a link. There were no more links on the page, and we need to do something else, and I will leave blank what we need to do here. So, this is all we need for the code to print_all_links. There are 2 parts left for you to do as the quiz here. See if you can figure out what should go as the test condition for the while and in the block for the else. And if that's correct, you'll be able to print all the links on the page.

Answer:

So here's the code that we need to finish. We need a test condition for the while, and in this case we really want to keep on going forever until we're done. So, we're going to use while True and then use break to stop the loop. The test condition is True, and the way we know when we're done is when the value returned as the URL was none. That means we got to the else, so to finish, we need to finish the else block by using break. Now let's test our code. We'll call print_all_links with our test string that has test 1, test 2, and test 3 as the 3 links. And when we run this, what we see is it prints out the 3 links, test 1, test 2, and test 3. We'll try a more interesting test. For a more interesting example, let's go back to the web page we looked at earlier. This was the page with the flying Python comic. And it has many

links in it, and we saw when we did view source we can see what those links look like, so the first link that we see on the page.

So if we use Command-F [or Ctrl-F], we can search for the first link on the page. We see that we have these links. The first one is the link to Archive, and that corresponds to what we see here at the top of the page. Here's the link to Archive, the link to Blag, the link to the Store and About and Forums. And so there's a lot of links on this page. If we go back to our print_all_links code, we can try to print them all. First, let's look at what we actually see when we do get_page passing in the URL of xkcd.com/353, which is the page we were looking at. And when we run this, we see a lot of text printed out, and that's exactly what we saw from view source, but now that's what we're getting as the result from get_page. That's the text of the web page. Not very easy to look at. Instead of using get_page, we'll print all the links on the page instead of just printing the whole page out. Now we're using our print_all_links procedure to print all the links in the xkcd page. And when we run this, we indeed see all the links on the page. At least we see most of them.

There are a few that we missed, and we'll talk about that in a later unit, but we're seeing many of the links on the page, including the first ones that we saw with the Archive and the Blog and the Store link, and we're seeing lots of other links. And you can see that we go all the way to the Buttercup Festival link and the license link. Those are the 2 at the bottom of the page. This is the link to license, which was the last one that we printed out. So, congratulations are in order. You've made it to the end of Unit 2, and I hope you understand the main concepts that we've seen so far. In Unit 1 we saw variables. We learned about programs. We saw how to do arithmetic. In Unit 2 we learned about making procedures, and we learned how to use if to make decisions, and we saw that those by themselves were enough to do every possible computer program as Alan Turing showed in the 1930s. We also learned about while loops, a way to make it more convenient to do things over again. And now we've actually got a really good start on our search engine. We can print all the links on the page. We still don't have our web crawler. We need to actually collect those links and do something with them. That's what we're going to do in Unit 3, and then in Unit 4, 5, and 6, we'll see how to use the corpus that we've built to do useful web searches, but we've come a long way, and I hope we'll see you back soon for Unit 3.