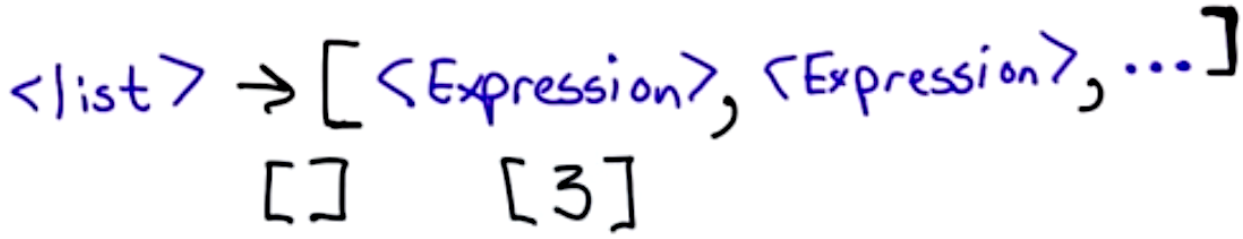
Intro to Computer Science

Lesson 03 Notes

How to Manage Data

Introduction

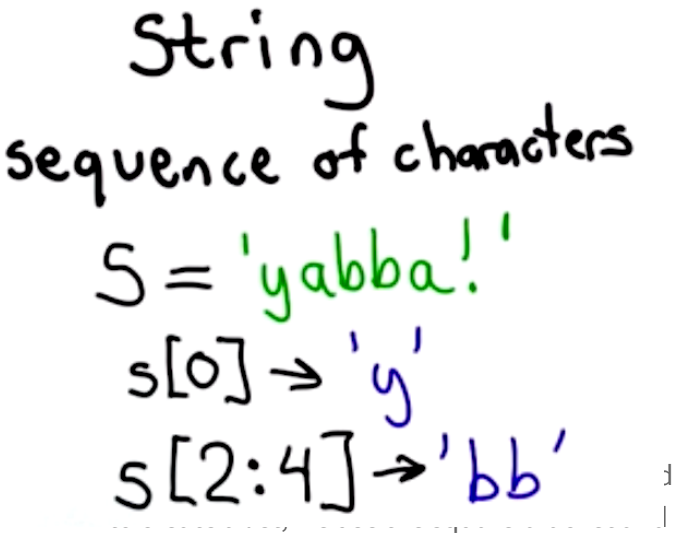
Welcome back for Unit 3. This unit introduces the next big idea we need for a web crawler, which is structured data. And by the end of this unit you will have finished building a working web crawler. The closest thing we've seen so far to structured data is the string data type introduced in Unit 1 and used in many of the procedures in Unit 2. A string is a kind of structured that's because we can break it down into its characters. The string has a sequence of characters, and we can operate o sequences of the string. What we could do with strings was somewhat limited, though, because the only thing we can put in a string is a character. Today, we're going to introduce the list data type, and lists are much more powerful than strings, so whereas for a string, all of the elements had to be characters, in a list, the elements can be anything we want. They could be characters. They could be strings. They could be numbers. They could also be other lists. Let's look at an example. When we created a string, we just put a sequence of characters surrounded by either single or double quotes. Here's an example of a string, and we could store that string in a variable by using an assignment. With a list, instead of using quotes to identify the list we use square brackets, and the elements are separated by commas. And just like with a string, we can assign the list that we created to a variable, so we'll store that list in the variable "p." With a string, we could use the square brackets to select elements, and when we index element 0, we'll get the first element of the string, a sequence of that character, which is the character "y." With lists, we can also use square brackets to access elements, so if we do p[0], that will evaluate to the first element of p, which is the string containing the single letter y. With strings, we saw that we could use the colon inside the square brackets to select a sub string of more than 1 character. Here we're selecting from position 2 through position 4. That will give us the third and fourth characters of the string, which is the sub sequence, the string "bb." We can do the same thing with lists. We can select from position 2 to position 4, but instead of returning a string, it will return a list containing those elements. It will give us a list of the third and fourth element of the variable p, which is the list that we have here. The general grammar for constructing a list is to have a square bracket followed by a list of any number of expressions by commas. We could create a list using just 2 brackets, a left bracket and a right bracket, and this would create a list containing 0 elements, also known as the empty list. We could create a list containing 1 element. That would be the square brackets with



Here we've created a list containing just 1 element, which is the number 3. Or we could create a list with many elements, as we did in the first example, where we have all of the strings separated by commas.

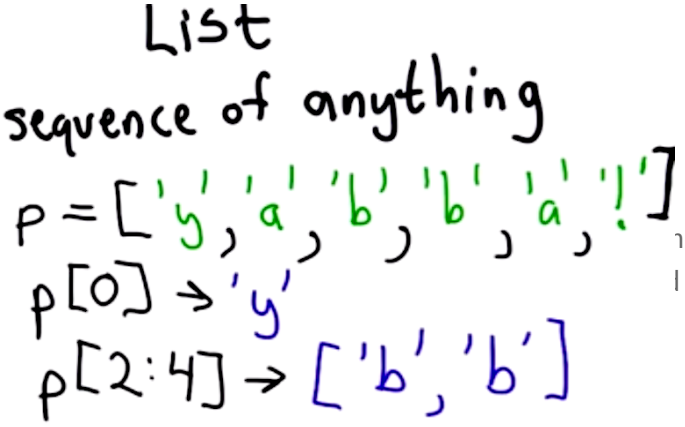
Programming Quiz: Stooges

So now it's time for a quiz to see how well you understand lists. Your goal is to define a variable with the name "stooges" whose value is a list of the names of the Three Stooges. If you're rusty on who the Three Stooges were, the names of the Three Stooges were Moe, Larry, and Curly. And remember that a list is ordered, so it's important that those values are in the list in the correct order, which is Moe first, Larry second, and Curly third.



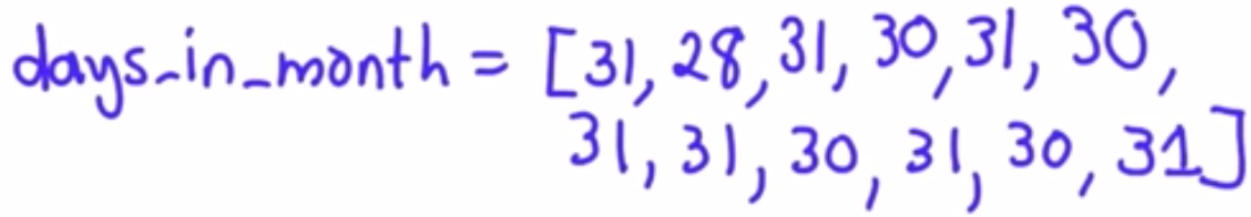
Answer:

put the elements that we want in the list between the brackets, separated by commas. In this case, we have the elements the string Moe, the string Larry, and the string Curly. To store that in a variable, we need an assignment statement. Now I know some of you are worried that in some of the Three Stooges movies, Shemp replaced Curly, and we'll see that we can actually do that with
the Python interpreter. And we can
it. It is a list containing those 3 strings.



Programming Quiz: Days In A Month

For this quiz, the goal is to see that you understand lists and can define a procedure that uses them. You'll be given a variable, days_in_month, which is initialized to a list containing 12 entries where each entry is the number of days in the corresponding month, so January has 31 days, so that's why we have 31 here. February--at least in a non-leap year--has 28 days. March has 31, and so on up to December that has 31 days.



Your goal is to define a procedure called how_many_days-- with underscores between the words-- that takes as input a number representing a month and outputs the number of days in that month, and we should use the conventional way of numbering the months, so here's a few examples. If we call how_many_days, passing in 1, meaning the month January, the output should be 31 since there are 31 days in January, and that corresponds to the entry in the zeroth position of days_in_month. If we call how_many_days, passing in 9, that's September. We should get 30.

Answer:

So here's one way to define how many days. We have our initial definition of days_in_month. We're going to define a procedure, how_many_days. And it takes one number's input. We'll call that the month. And what we want to do is return the number of days in that month. We can find that by indexing into the days_in_month list, and we want the element that corresponds to the month that was passed in. That isn't the element of the month directly because elements in the list start from 0, months start from 1. We need to subtract 1 from month to get the correct element. Let's try that. We'll print the number of days in month 1, which is

January. And we run, and we get 31. Let's try month 9, which is September, and we run and get 3. What happens if we pass in, say, a number that's not a valid month? Let's say we pass in 13. Now we run this, and what we get is an error, and you can see what the error is. It's complaining that the list index is out of range. That means we've tried to find an element in a list that doesn't exist, and we did it exactly at this point where we're returning days_in_month of month - 1, and the value of month was 13. There's no element at position 12 because days_in_month only has 12 elements.

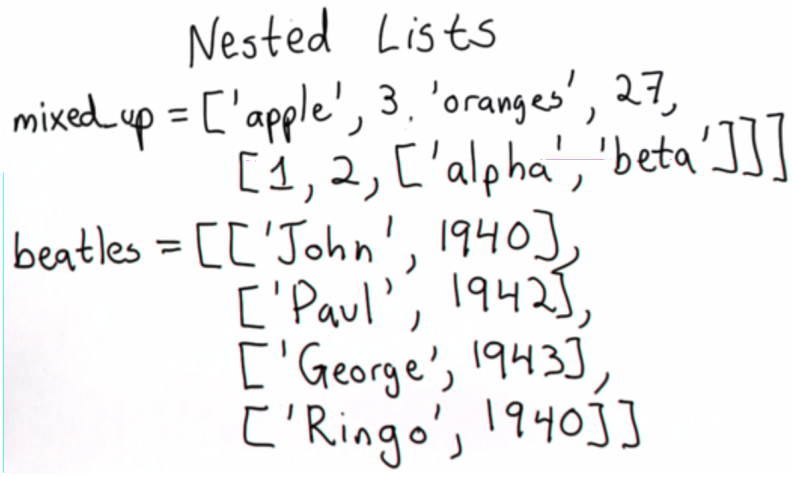
```
2 days_in_month = [31, 28, 31, 30, 31, 30,
3                  31, 31, 30, 31, 30, 31]
4
5 def how_many_days(month):
6     return days_in_month[month - 1]
7
8 print how_many_days(13)
9
```

```
Traceback (most recent call last):
  File "/code/knowvm/input/test.py", line 8, in <module>
    print how_many_days(13)
  File "/code/knowvm/input/test.py", line 6, in how_many_days
    return days_in_month[month - 1]
IndexError: list index out of range
```

Nested Lists

So all the examples we've seen so far, all of the elements in the list had the same type. We had a list where all the elements were strings. We had a list where all the elements were numbers. There's no restriction on lists, though. We can make all of the elements any type we want. We can mix and match different things in 1 list, so here's an example. We've put some strings and some numbers together in 1 list. That's perfectly okay. Where it gets even more interesting is we can have a list inside a list. Now one of the elements of "mixed up" is itself a list, and we can have another list inside that. Here we've defined a new variable, mixed up. It contains a string, a number, a string, a number, and then a list. The fifth element is itself a list that contains a number, a number, and contains another list. This example shows the versatility of lists, that we can put anything we want in them. It doesn't seem particularly useful because there's no structure to the things we've put in the list. So let's try another example that may give a better idea of why it's so useful to be able to have

lists inside lists. Before we defined the variable "beatles" as a list of 4 strings, just the names of the Beatles. Now we've defined the variable beatles as a list of lists where each element of the list is a list of the name of the person as well as the date when he was born. We have the first element in our list as John, who was born in 1940. Let's see how that works in the Python interpreter. When we start to have long lists, especially lists of lists like this, they might not all fit on 1 line. It's okay to divide things into 2 lines, but we have to be a little careful where we do the division to make it clear to the Python interpreter that it's all part of 1 list. The place where you want to do the separation is after a comma, and if you indent things in a way that makes that clear, then you can see the structure, that these 2 lines are still part of the same list, and it's the closed bracket here that closes the list. Now let's try printing that out, and we see the list containing 4 elements. Each element itself is a list containing 2 elements. We can print just 1 of the elements. We'll print the element at position 3, which is actually the fourth element of the list. We see that element itself is a pair. That's a list. So if we want to get just the name from that, well, then we need to use index again, so we can use index again to get the 0 position element of that list, and that will give us just the name. If we used index 1 here, that will give us the year.




```
1
2
3 beatles = [['John', 1940], ['Paul', 1942],
4              ['George', 1943], ['Ringo', 1940]]
5
6 print beatles[3][1]
7
```

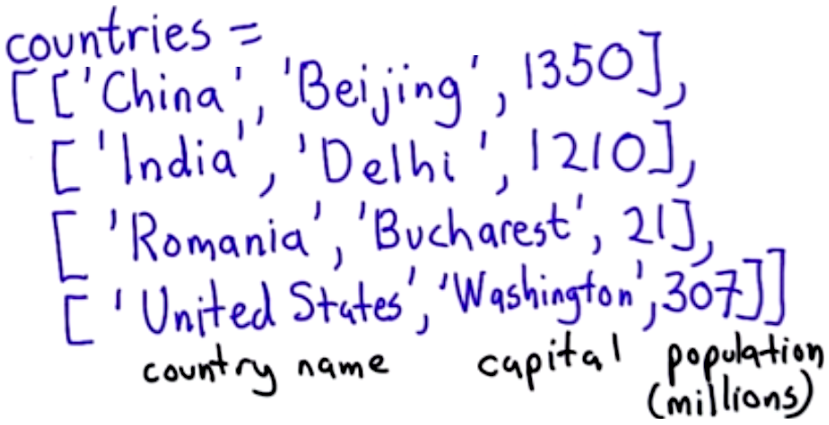
```
1940
```

Programming Quiz: Countries

So now we're ready for a quiz to see if you understand nested lists. For this quiz, you're given a variable named "countries," and it's defined with a list of information about some of my favorite countries. And for each country, there's a list containing 3 things: the name of the country, the capital, and its population in millions. I should point out, of course, for those of you in Romania or in smaller

countries, that when it comes to countries, bigger is not always better. First, try to write code that will print out the capital of India. We can find India as the second country in our countries list. And the capital is the second element of that list. You obviously could solve this by just printing out "Delhi." I don't want you to solve it that way. I want you to solve it by finding an expression that uses the countries variable to find the right value.

Answer:



So the answer is we need to find the second element of countries, so that's this element. Remember that lists are indexed starting from 0, so to find the entry corresponding to India, we need to find the entry at position 1. The result of countries[1] will be the element of countries at that position, which is the list containing 3 elements, the string India, the string Delhi, and the value 1210, which is the population of 1.2 billion people. To select the capital, that's the second element of that list, so we use index 1 again to get that element, and the value of countries[1][1] will be Delhi. And then we can just use print to print that out. Let's try that in the interpreter. So, running countries[1] will give us the element of countries that corresponds to India. And then to get the capital, we select element 1 from that.

```
3 countries = [['China', 'Beijing', 1350],
4               ['India', 'Delhi', 1210],
5               ['Romania', 'Bucharest', 21],
6               ['United States', 'Washington', 309]]
7
8 print countries[1][1]
9
```

```
Delhi
```

Programming Quiz: Relative Size

So for the second part of this quiz, the question is what multiple of Romania's population is the population of China? To solve this, you need to divide the population of China, which you can extract from the countries variable, by the population of Romania, which you can also extract from the countries variable.

Answer:

To get the entry for China, we need to select the first element of countries that's indexed at position 0. To get the population, well, that's the third element of that list, so that's the entry at position 2. To compute the multiple, we need to divide that by the same value for the entry for Romania, and when we run this, we see that the population of China is approximately 64 times the population of Romania, and I should point out that this is integer division, so this is not the exact result. If we looked at actually multiplying 21 by 64, the result that we get is 1344, so 1 billion 344 million. But the population that we get for China was 1 billion 350 million. If we wanted a more exact answer, we need to use floating point division. We could do that by changing the values in our list to make one a floating point number. Adding a decimal point would be enough. Now when we run it, we get a more precise answer. Of course, these populations are just estimates, so the answer of 64 would probably be a better answer to the actual question.

Run

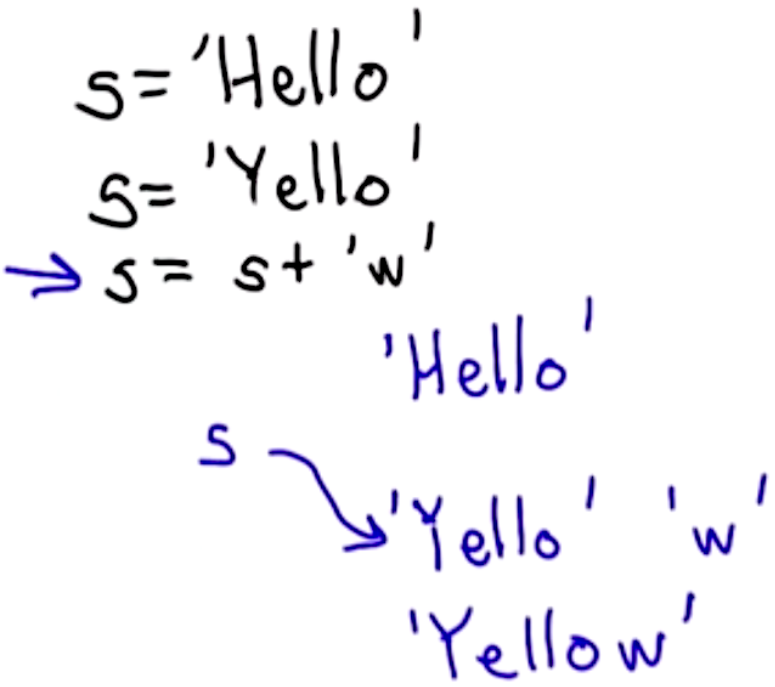
```
1
2
3 countries = [['China', 'Beijing', 1350],
4              ['India', 'Delhi', 1210],
5              ['Romania', 'Bucharest', 21.],
6              ['United States', 'Washington', 309]]
7
8 print countries[0][2] / countries[2][2]
9
10
```

```
64.2857142857
```

Mutation

So I said there were 2 main differences between strings and lists. The first we've seen already is that a list can hold anything we want, whereas a string, the elements could only be characters. The second big difference between lists and strings is that lists support mutation. Mutation sounds a little bit scary, and in some ways it is scary. It makes it much harder to understand what our programs mean, but it's also very powerful. What mutation means is we can change the value of a list after we've created it. To explain what that means and to see why it's so different from the behavior we can get with strings, let's recap what happens with strings. We can create a string. We can store it in a variable. We can change the value of

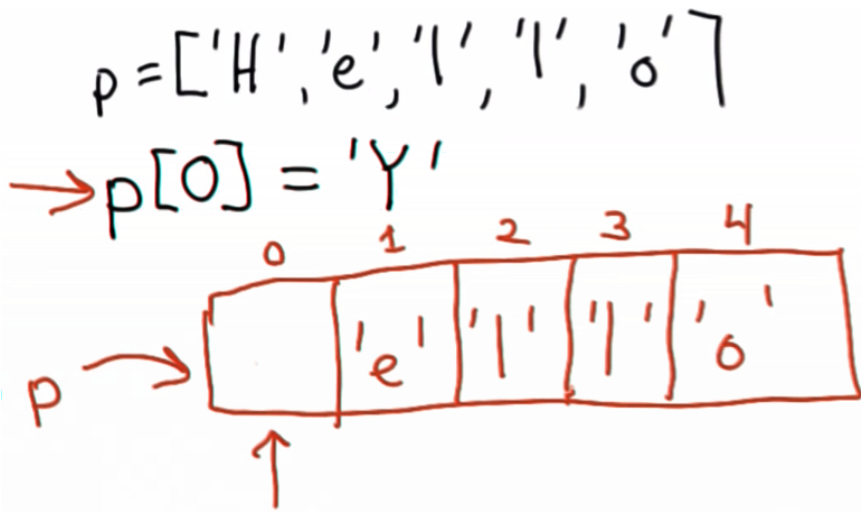
the variable by creating a new string. So what's going on there, it's changing the value of s, but it's not changing the value of the string. When we did the first statement, this one, it created a string with the value "Hello," and it introduced a variable, s, which referred to that string. When we execute the second statement, that creates a new string with the value "Yello," and it changes s to refer to that new string. It hasn't changed the string that we created with the value Hello. We just don't have a way to refer to that any more. With strings, we can also use the + operator to concatenate 2 strings. It looks like this is changing the value of the string. You might think that the result of this is, well, now the string that was previously Yello spelled wrong is now Yellow spelled with the w added at the end. But that's not actually what happens. We don't actually modify the string we had. What happens when we use the + operator, it creates a completely new string where the value of the new string is the result of concatenating the 2 input strings. When we execute the third statement, first a new string with the value w is created. Then when we do the concatenation, it doesn't modify the previous string that we've created. It creates a whole new string which will have the value "Yellow" spelled with a w. And then the assignment changes what s refers to to refer to the new string. Now let's see what we can do with lists.



A List Of Strings

We've created a list using the square brackets that contains 5 strings. Each one is a 1-letter string spelling out the word "Hello." And we've stored that in the variable p. We could do the same thing we did with strings. We could reassign p to a new value. And if we did that, that would create a new list containing the letters Y-E-L-L-O and assign that to the variable p. What we're going to do instead is mutate the list that we already created, and we can do that by modifying the value of its elements. Here we have an assignment where on the left side of the assignment we have p[0]. This means select from the list that p refers to. Find position 0. Instead of getting the value at that position, though, since it's on the left side of an assignment now, we're going to replace that value with what's on the right side of the

assignment, in this case, the string containing the single letter y. Let's see what that looks like. When we created the list initially, we had a list like this, and I'm going to draw the list showing boxes. This is a list. Each box is 1 element of the list, and after the initial assignment, the value p refers to is this list. Now when we do the assignment where p[0] is on the left side, well, p[0] will find this position. The assignment will replace what's there. So, replace the H with the Y. Note that we did not create a new list. P still refers to the same list. We didn't have to change the arrow. But we've changed the value of that list. Let's try this in the Python interpreter. I've created a list with the elements being 5 strings spelling out the word "Hello." And I can see that that's the value of p. Now I'm going to use the assignment expression to replace the value in the first element and print p again. Now when we run, we see that for the first print the value of p is Hello. For the second print, the value of p is Yellow. We can change other values the same way, so now we've changed the value in position 4 to an exclamation point, and we can see the third line printed out. We have "Yell!"



Programming Quiz: Different Stooges

So now it's time for the first quiz about mutation. In a previous quiz, we defined the variable stooges to hold 3 strings, strings Moe, Larry and Curly. But in some of the Stooges films, Curly was replaced by Shemp, so your goal for this quiz is to write 1 line of code that changes the value of stooges to be the list containing 3 strings, Moe, Larry, and Shemp, but doesn't create any new list object.

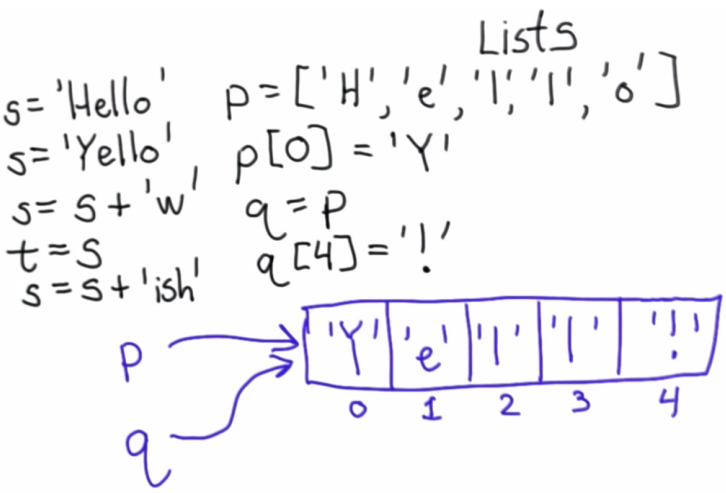
Answer:

The answer is we can do this using mutation. We just need one assignment that replaces the value in position 2 of stooges with the string "Shemp."

Yello Mutation

So far I've told you that mutation modifies the existing object. But you can't really see the difference from what we've done with strings. Where we'll really see the difference is when we introduce a new variable. Let's go back to the example we

had before, and now we're going to add an extra assignment statement, and we'll introduce a new variable. Suppose we introduce the variable "q." And we assign p to q, and that means the value of p, which is the object that's this list, is now what q will refer to. The important thing here is after the assignment, p and q refer to the same list. Suppose we did an assignment statement to modify the value of one of the elements of q. Well, that will change element 4 of q, so this is element 4 of q. It will change that value to the new exclamation point. It also changed the value of p. Even though the assignment statement didn't include p, the fact that p and q refer to the same object means that it changed the value of p. To show you that things are different with strings, let's try that with a string. See if you can guess what happens when we try to use assignment to replace the first letter in the string. Let's run it to see what happens. And what we get is an error, and we get an error because the string is not mutable. There's no way to change the value of the string, and the error says there's no way to do assignment in a string, that that type of object, because it's immutable, does not support assignment. A key difference between mutable objects and immutable objects is once an object is mutable, then we have to worry about other variables that might refer to the same object. We can change the value of that object, and it affects not just the variable that we think we changed, it affects the value of other variables as well. Let's see an example of that. I've initialized p to the list containing the strings "Hello." Now I have an assignment that introduces the new variable q and assigns p to that variable. And we'll print out the values of p and q, and we'll see that both p and q contain the string Hello. But now let's change the value at position 0. Now we have an assignment that stores in the value at position 0 of p the letter y. This changes the value of p. What may be more surprising is this also changes the value of q. Even though we didn't use q in the assignment, it changed the value of q because q refers to the same object as p.



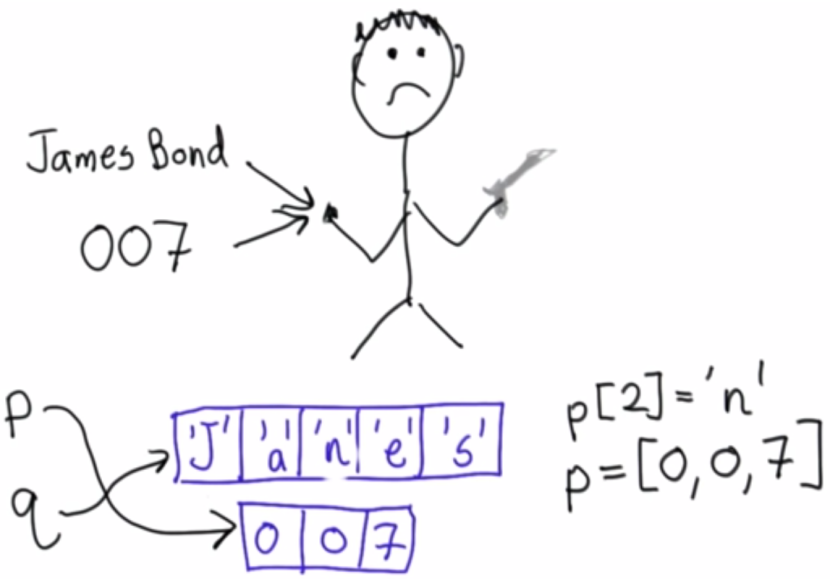
Run

```
1
2
3 p = ['H', 'e', 'l', 'l', 'o']
4 q = p
5 p[0] = 'Y'
6 print q
```

```
['Y', 'e', 'l', 'l', 'o']
```

Aliasing

When we have 2 different ways to refer to the same object, that's called aliasing. Aliasing in programs is similar to the way we think about aliasing in Spycraft. You might not be able to tell from my drawing, but this is James Bond. And we can refer to James Bond either by his name, James Bond, or by his spy name, 007. Both of those are ways to refer to the same person. If something happens that changes the state of James Bond, so instead of pleasantly enjoying a martini, like he is now, the bad guys are after him, and he grabs his gun and is no longer so happy, well, that affects the state of both James Bond and 007. Both names refer to the same person, so whatever state that person has is true for both names. Aliasing for variable names has the same property, so if we have 2 variable names that refer to the same object, any change we make to the object p refers to also affects the value that q refers to. If we did an assignment using p to change the value in the second position on p, that also changes the value the name q refers to. If we do an assignment like this where instead of changing the value of one of the cells that's part of the p object, we assign p to a new value, well, this assignment we create a new object, a list containing 3 elements, the numbers 007. And it will change what the variable p refers to to point to that new object. The value of q is the same as it was before. It still points to the same object, which we haven't modified.



Programming Quiz: Secret Agent Man

So, now we're ready for a quiz about aliasing. The question is what is the value of agent index 2 after the following code runs? And we have 3 assignment statements. The first assignment statement creates the list 0, 0, 7, and assigns that to the variable named spy. The second assignment assigns spy to the variable named agent, and then the third assignment assigns agent index 2 +1 to the position at spy index 2. So what's the value of agent 2 after all of this code?

Answer:

The answer is 8. Let's try this in the Python interpreter to see why. These are the first 2 assignments, and after these assignments we can see that agent has the same value as spy. Both spy and agent have the value a list containing 3 elements. The element values are 0, 0, 7, and they're actually the same list object because both agent and spy refer to that same list that we created in the first assignment. This means the value in agent position 2 is the number 7. When we do the third assignment we're assigning into position 2 of spy the value at position 2 of agent plus 1. The value at position 2 of agent is 7, add 1 to that we get 8, and we put that in position 2 of spy. Since agent and spy refer to the same object that changes the value of both spy and agent, and we can see that both agent and spy now contain the value 8, and position 2.

Programming Quiz: Replace Spy

For this quiz your goal is to define a procedure named replace_spy that takes as its input a list of 3 numbers and modifies the value of the third element in input list to be 1 more than its previous value. To solve this quiz you're going to have to experiment and understand how values are past procedures in Python. Something we haven't talked about yet, but if you solve this quiz you'll have a good understanding of how things might work. The behavior that we want is shown in an example, so if the value of spy is the list 0, 0, 7, we call the procedure replace spy, passing in spy. Note that we're not using the result. We're not assigning this back into a variable spy. We're just calling replace spy, but after the call the value that spy refers to has changed. Now it has the value 0, 0, 8. So, I haven't yet told you what it means to pass a mutable object like a list to a procedure, but try to define replace spy and see if you can figure out what happens.

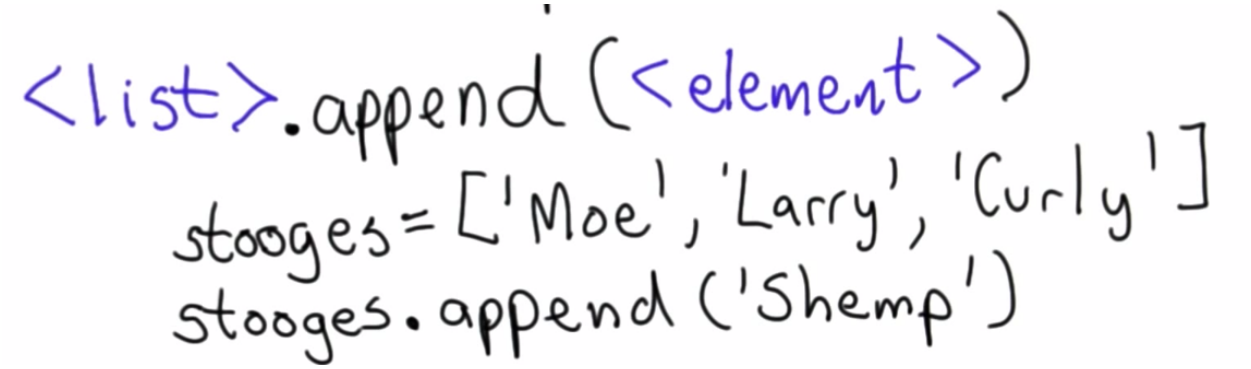
Answer:

Here's the answer: we can define a procedure named replace spy. It takes 1 parameter as its input. I'll call the parameter P. We could call it spy and that might be a more natural name for it here, but to avoid confusion with the variable spy, we're using in the example, I'll call it P. For the body of replace spy what we want to do is change the value in position 2 of the value past it. So, we can do that within an assignment statement. On the left side we're selecting the cell, and we're replacing it with a value that we get from the old value in that position plus 1. This has the behavior that we want, and to have a better understanding why let's see what happens when we pass a value to our procedure. Here's what we had before the call. We've initialized the variable spy to hold the list containing 3 elements that are numbers 0, 0, 7. Then we make the call to replace spy, passing in spy. What happens when we call a procedure is that the name of the variable in the procedure now refers to the object that's passed in. So, now the value of P of the parameter here refers to that object. Then we do the assignment. We replace the value in position 2 of P with it's previous value plus 1, so that changes this value to 8. The parameter P and the variable spy both refer to the same object, so that also changes the value of spy. Not that there's no return statement here. We don't need to return a new value.

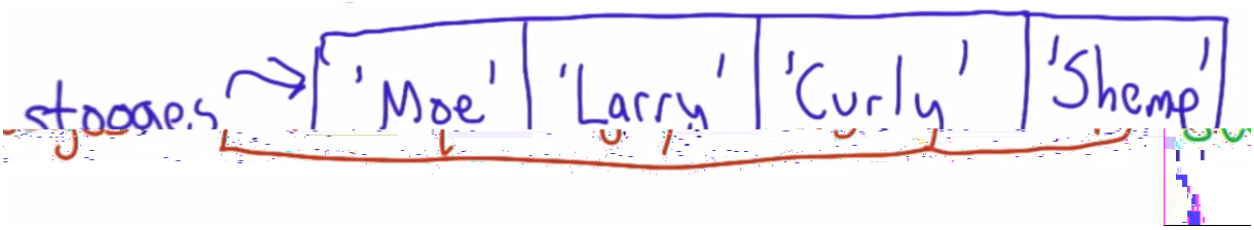
We've modified the value of the list that was passed in, and I want to emphasize that this is different from what's happened in other procedures. If you remember in an earlier exercise you defined the procedure INC. It takes 1 input, we'll call it N, and it adds 1 to that. If we remove the return statement from INC, it doesn't change the value that's passed in. It does nothing, and here's why: suppose we had code like this, we'll introduce a variable A, give it the value 3; we'll call INC on A. What happens here--well, the variable A that name refers to the value 3. When we passed that to INC--well, the same thing happens as happened when we passed the mutable list as parameter P to replace by. The name N now refers to the same value as A referred to, but we can't mutate that value. What happens inside INC is an assignment statement. We assign to N the value N +1, so that the changes the value N refers to. Now N refers to a value 4, but that doesn't change the value A refers to. A still has the value 3. So, we can see with numbers we can't mutate the value of a number. If we could everywhere the value 3 existed before--well, now it would mean 4. That would be pretty confusing.

List Operations

I hope you're getting a sense that lists are very powerful. That by using mutation, by having lists that could contain other lists or any other kind of data we want, we can build very complex data structures, and we can do lots of interesting things. I'm going to introduce 1 more list operation, which will allow us to add a new element at the end of the list. We've seen that we can use lists to store complex data, that elements of the list can be any type we want including other lists, and we've seen that we can use mutation to change the value of a list, and that mutation is visible through any reference to the same list object.



Now I'm going to introduce some other list operations. The first one is append, and append is like a procedure but it's a method, so we use it similar to the way we use to find on strings. We'll have a list first, then a dot followed by append, and what we pass in is the element we want to append to the list. Append will add a new element to the end of a list, and the important thing about append is that it's mutating the list that its invoked on.

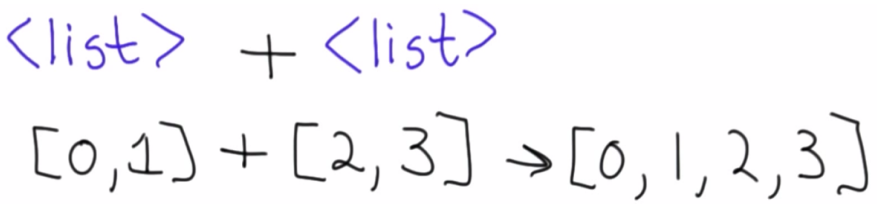


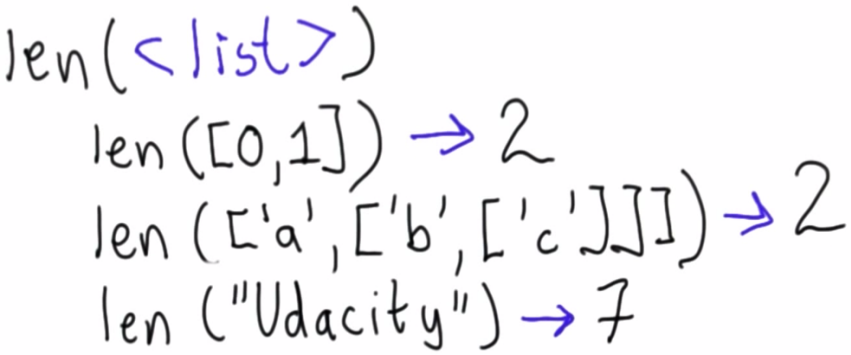
It's not creating a new list; it's mutating the old list. As an example of the use of append let's assume that instead of replacing curly in the 3 stooges, we want to end up with 4 stooges. We'll add Shemp and add the other 3 as they are. So what we want to do is to append Shemp at the end of the list we have. We would do that by invoking append on the stooges, passing in the string Shemp as the input. Here's what happens after the first assignment, the name stooges refers to the list containing the 3 elements, Moe, Larry, and Curly. When we invoke append it modifies that object, adding a new element to it. After the append, the list that stooges refers to now has 4 elements. We have not created a new list. Note that there's no assignment from the result of append. What we've done is modify the value that stooges refers to by adding a new element to it.

List Addition And Length

That's what append does. We're going to introduce 2 other list operations. The next one is plus. Plus behaves very similar to the concatenation operation for strings. If we have the list 0, 1 + the list 2, 3 the result is the list 0, 1, 2, 3.
What plus does like
concatenation for strings is
it produces a new list. It
doesn't mutate either of the
input lists. I want to
introduce 1 other operator
that works on lists, and that's the len operator. Len is short for length, and we use len like this: it looks like a procedure call. We pass into len the object that we want to know the length of that can be a list. Len actually works for many things other than lists. It also works for strings. It works for any object that's a collection of things, and the output from len is the number of elements in the input. For example, the result of len applied to the list 0, 1 is 2, since there are 2 elements in the list. The result of applying len to this list is also 2. It looks like there are many more elements here, but len is only

counting the outer elements. If 1 of the elements of a list is a list, it doesn't matter how many elements that list contains. It only contributes 1 to the length of the original list, so the result of this call would also be 2. We can also use len on a string, and the output will be the number of characters in the string. In this case the string Udacity has 7 characters, so the output is 7. Now we are ready for some quizzes to see how well you understand the 3 list operations we've introduced. The append operation which is invoked on a list and takes an element as a parameter and adds that element to the end of the list. The plus operator which operates on 2 lists as its operands and produces a new list that consists of all the elements in the 2 lists put together. And finally the len operator that takes a list as its input and produces an output number that is the length of the list. The number of elements in the input list.





Programming Quiz: Len Quiz

Now we're ready for our first quiz about append, plus, and len. The question is to answer what is the value of len of P-- --the length of P after running the following code? These are the 3 statements: the first one creates a list 1, 2 and assigns that to the variable P. Then we use append, passing in 3 as the value to append, invoking append on the variable P, and then we have an assignment that assigns the P the result of P + the list 4, 5. The question is after running those 3 statements, what is the value of len of P?

Answer:

The answer is 5. Let's step through each statement to see why. The first statement creates a new list. It's elements are 1 and 2, and makes the variable P refer to that list. The next statement evokes append on P, passing in the value 3. Append adds an element to the end of the list, so after this the value that P refers to will include 1 extra element with the value 3. The next statement uses the plus operator. The operands to the plus are P, which we already have, and the new list containing the

elements 4 and 5, and remember what happens with plus; it doesn't mutate any of the inputs. It creates a new list that consists of all the elements concatenated together. That will be this 5-element list. We assign that to the variable P, so now P will no longer refer to the list 1, 2, 3. It will refer to this new list that has 5 elements, and at the end of this, the len of P is 5, since there are 5 elements in P.

Programmings Quiz: Append Quiz

For this quiz we're asking for the value of the length of P after running these 3 statements. This is a little trickier than the previous quiz. Try to see if you can think about the answer and find it yourself. Of course it's okay if you want to run the code in the Python interpreter as well, but see if you can figure out the answer without doing that.

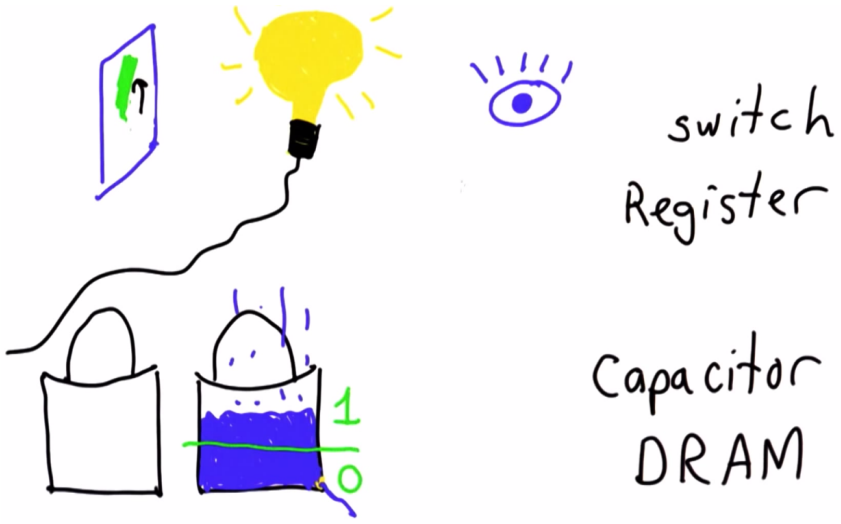
Answer:

The answer for this one may be a little surprising. The answer is 3, here's why: the first statement creates the list with the elements 1, 2, and P refers to that list. The second statement creates the list with the elements 3, 4 and assigns that to the variable Q: The third statement appends a new element on the list P. That element is the list with 2 elements, but what actually happens is not adding 2 elements to P. What happens with append is we're just appending 1 new element. That element happens to be a list. The way this looks is we have a new element in P. The value of that element is actually the list object that Q refers to. That's why the length of P is 3. Let's try this in the Python interpreter. Here we see the list that's created. We can see that the length of that list is 3. What do you think will happen if we try an assignment like this? We're replacing the element at position 1 of Q with the value 5. Does that change the value of P? It does, and if you remember the picture we drew, it's not surprising that it does. Remember that the object we had created, the third element of P, is actually a reference to this object containing the elements 3, 4, which is the same list object that Q references. When we change the value in position 1 of Q that also changes the value of P.

How Computers Store Data

Now you understand a little about how to use a complex data structure like a list. I'm going to talk some about how computers actually store data. To use a list you don't really need to understand this, but it will certainly give you a better appreciation of what's going on inside the computer. In order to store data you need 2 things: you need something that preserves state and can be in more than 1 state, One way to think about that is to think about a light switch. Here we have a switch; we have our light bulb, presumably it's
also connected to some
power source, and when we
flip the switch that turns the

light bulb on. It changes its state. The light bulb has 2 different states. It can be off or on, so that means it can store 1 bit of data. In the next lecture I'll talk more about what a bit means, but something that can have 2 distinct states can store 1 bit, and if we had enough light bulbs well we could store many, many bits, which would be enough to keep track of any amount of data that we wanted. We don't need just something that can change state though. We need something that can also sense the state. With a light button, well, that could be someone's eye. You can see if the light bulb is on or off, and then you know its state. You could have a light sensor that would do that if you wanted to make that automatic. The way computers store data is sort of like this, except for it's using much less energy and much less space than a light bulb to store the state of data, and it's not using a human eye. It's using sensing electrons or sensing magnetism to sense the state of something. That's 1 way to store data is like a switch, and the fastest memory in your computer works that way. The data that's stored directly in the processor, which is called the register, is stored like a switch that makes it very fast to change its state, to record what the state is, but it's also like a light bulb. It means that when we turn the power off we lose the state, so it's not persistent, and it's a lot more expensive than other ways of storing data. The other way computers store data is more like a bucket. We could represent having a 1 by having a bucket that's full. We could represent having a 0 by having a bucket that's empty.



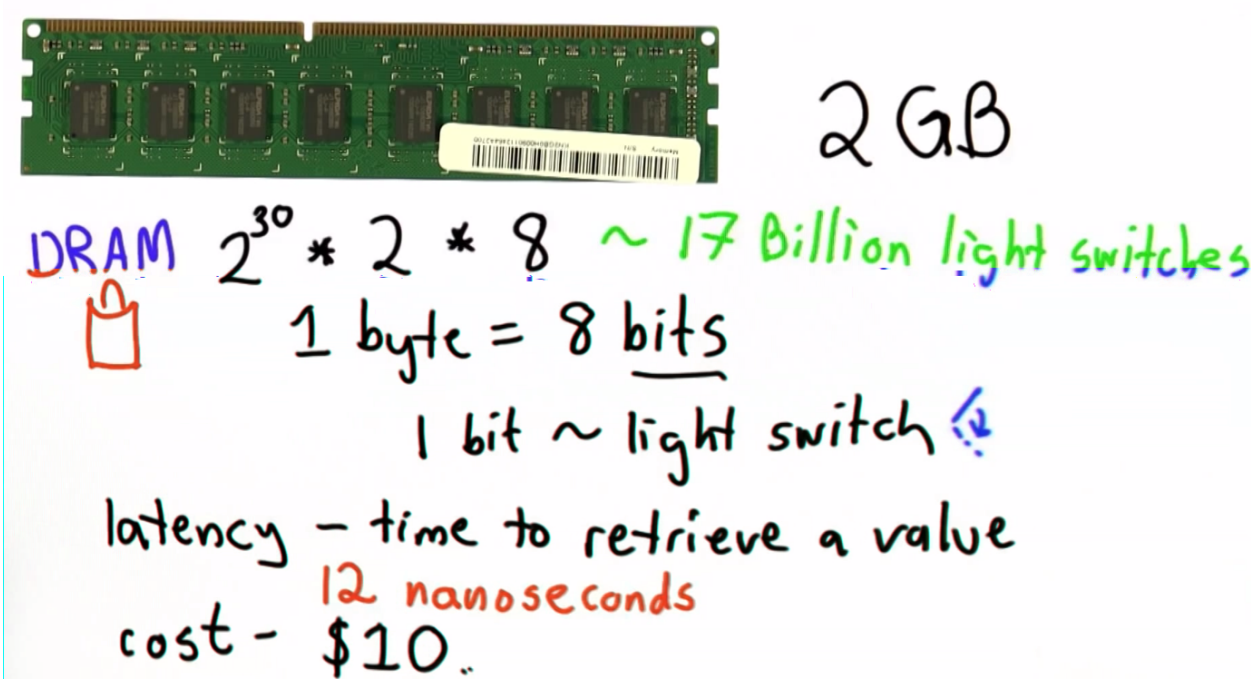
Then to check the state of the bucket, we could weigh the bucket or look at the bucket to figure out whether it's full or empty, and the difference between buckets and light bulbs is-- --well, buckets leak a little bit. Water is falling out of the bucket and they also evaporate. If we want to store data using a bucket it won't last forever. Eventually the bucket will evaporate, and we can't tell the difference between a 0 and a 1, and I should also point out that the amount of water in the bucket--well, we're going to have to decide some threshold where if it's above that threshold we think that it represents a 1. If it's below that threshold we think it represents a 0. We could certainly imagine having many more levels than just 2 in the bucket. For computing it's almost always the case that we want to use a digital abstraction where everything only represents 1 of 2 things. That makes it less likely that we get it wrong. As the water evaporates we know it's still a 1 until it gets really near the bottom, but when it gets near the bottom there's going to be some left, and we want to know when it becomes a 0. When we're doing this with electrons we don't call it a bucket. We call it a capacitor. The memory in your

computer that works this way is called DRAM.

Dram

I want to give you a better idea what these actually look like, and my wife was kind enough to let me take some parts out of her computer. Actually, she took them out of the computer so I could show you what they look like. This is DRAM; this is 2 GB of DRAM, and what a gigabyte means is approximately a billion bytes. What it actually means I'll show you in the Python interpreter. In Python we can do exponentiation by using a star star. This is what we would write mathematically as 2 to the power 10. So 2 to the power 10 is 1024. This is called the kilobyte; we use the prefixes like the thousand prefixes, but the actual values in computing are usually powers of 2 instead. They're a little bit different than 1000. It's actually 1024 when we talk about a kilobyte, and a megabyte is 1024 KB. That's equivalent to 2 to the power 20. A megabyte is just over a million bytes. A gigabyte is 2 to the power of 30, which is just over a billion bytes, and a terabyte is 2 to the power of 40, which is just over a trillion bytes. These are the main units we're going to talk about when we think about memory.

Here we have 2 GB of memory, so that means we have We have 2 to the 30 times 2, which is 2 GB times 8, which is the number of bits in a byte, and 1 bit is the equivalent to a light switch. It can be in 1 of 2 states, so with 1 byte we have the equivalent of 8 light switches. This is comparable to having 17 billion light switches. The kind of memory that we have here is actually DRAM, so it's really more like having 17 billion buckets, and that means when you turn your computer off everything that's stored in the DRAM is lost.



To keep the memory here, it's like a bucket, but instead of storing water it's storing electrons. It's a capacitor and when the power goes off that memory is lost. There are many different types of memory inside your computer. I mentioned registers briefly, that's the fastest memory that is built right into the processor. The main things that matter that distinguish the memory that you have is the latency which is the time it takes to get a response. So the latency you can think of as the time to retrieve a value from memory.

We'll talk about latency more next class in terms of what that means in networks, which is really the same thing. For this DRAM the latency is about 12 nanoseconds, and as a reminder, a nanosecond is 1 billionth of a second. The other thing that really distinguishes memory, and the reason we have so many different types of it, is the cost. The cost varies depending on mostly the latency and the amount of memory you can store. The cost for this memory, if you bought it today, is about $10 for 2 GB. That's 10 US dollars, about 7 Euros.

Programming Quiz: Memory Hierarchy

Units like nanoseconds are fairly hard to understand, so what I want to do to give a better understanding of what the different types of memory in a computer are is turn those into different units and compare them. If we think about the light bulb switch, which certainly would not be a reasonable way to store memory in a computer today, the cost per bit is probably about 50 cents. So we can buy light bulbs for about 50 cents. Let's assume the sensing is free. It's already going to be ridiculously expensive. I'll call it US dollars, 50 cents per bit for using a light bulb and a light sensor. The latency is about 1 second. And what we're going to do in the quiz is convert latencies in terms of time in terms of the equivalent for the distance light travels in that amount of time. The other type of memory I mentioned-- and we didn't see this because it's sort of hard to extract from the computer-- is the CPU register. The registers are part of the processor, so it's hard to measure their actual cost. It's probably close to a 10th of a cent, though, if we look at the total cost of the processor and how that depends on the number of registers you have.

The amount of memory that you have in registers is close to a kilobyte or 2 in a typical processor. Only a few thousand bytes in the registers. The latency is the shortest of any of these. It's less than a nanosecond. And the time to get memory from a register has to be short enough that you can do it within a single cycle. So we saw our cycle speed for the processor was 2.7 gigahertz. That means we need to do 2.7 billion cycles in a second. The time of a cycle is less than 0.4 nanoseconds. Within a given cycle, we at least need to be able to get memory out of a register. So that's an approximation of the time there. We'll measure the latency distance in terms of meters for light. So the third type of memory we've talked about so far is the DRAM. The cost per bit for the DRAM--well, our DRAM costs about $10 and it stores 2 gigabytes. So in order to write down the cost per bit, we really need a new unit. If we try to write it in terms of dollars and cents, it's going to be far too hard to see. So to fit DRAM into this, DRAM costs $10 US dollars for 2 gigabytes. If we want to write that in terms of a meaningful

cost per bit, we need a new kind of currency. We're going to need far too many zeroes before we get to the value using regular dollars and cents.

So we're going to use a new kind of currency. To measure the cost of DRAM in a meaningful way, we need a new kind of currency, which we'll call a nanodollar. A nanodollar is one-billionth of a dollar. So if we have a billion nanodollars, that would make 1 dollar. I don't have one of these on paper. It's money that's really not worth the paper that it's printed on, but it's a useful unit for measuring the cost per bit. Now we have a more useful currency for measuring the cost per bit of DRAM. The cost of DRAM turns out to be, in terms of nanodollars, 0.58 nanodollars per bit. And we saw that the latency for DRAM was 12 nanoseconds. So now for the quiz I want you to figure out the latency distance for each of these different types of memory. Note that the units are different here. We used dollars and nanodollars. For the distances we're going to use kilometers, meters, and meters. You've got all the information you have here. As a reminder, the speed of light is about 300,000 kilometers per second.

Answer:

We can compute these using the Python interpreter. You didn't need to submit your code for this. You needed to compute the answers. And the answers that you should have gotten is that a light bulb, the latency distance is approximately 300,000 kilometers; for a CPU register, where the latency is less than 0.4 nanoseconds, the equivalent latency distance is 0.12 meters, about 120 centimeters; for the DRAM, DRAM is much cheaper. The latency is much higher than a CPU register. That corresponds to 3.6 meters of latency distance.

Programming Quiz: Hard Drives

There's one last type of memory in your computer that I want to talk about, which is the hard drive. This is what a hard drive looks like. You can see that this hard drive has 1 terabyte of data. If you turn it upside down, you can maybe get a little better sense of what's going on here. It's a little hard to see. My wife drew the line at opening the hard drive, so I'm sorry I can't show you inside it. But inside here there's a bunch of disks that are spinning, and the disks are storing data in a magnetic way, and there's a read head that can read data from the disk as well as write new data from the disk. So this is a very slow way of storing data compared to what we have in the DRAM, but it can store a lot more data with a lot less cost. There are 2 important properties that the hard drive has. The hard drive is a lot bigger than the DRAM. It stores a lot more memory. This was 2 gigabytes. This is 1 terabyte, so that's 500 times as much memory. And a terabyte is close to a trillion bytes, so that's 8 * 2 to the 40 bits, which is equivalent to 8.8 trillion bits.

This is enough memory to store about a quarter million songs. The latency for a hard drive is much worse than the latency for the DRAM memory. And the reason for that, if you think about what you

have to do to get data out of the hard drive, you've got these disks that are spinning in a physical thing, so you've got to wait until it spins to the point in the read head where you can read it. If it's not in the right place, the read head also has to move across this. You're moving real physical things, so that takes time. The latency for a hard drive is about 7 milliseconds, and a millisecond is 1/1000th of a second. And that's the average time. The time varies depending on where the data is. So if you have to wait for it to spin all the way around and move to the head, that can take a lot longer. If you're lucky, the data is in the right place and it doesn't take that long. The cost of this hard drive is about $100 US dollars. So let's add the hard drive to our data table. For the quiz it's up to you to figure out the values for the hard drive in the data table. As a reminder, we said it was $100 US for a 1.0 terabyte hard drive. Your goal is to figure out the number of bits. I've told you the latency is 7 milliseconds. That's a milli, not a nano. You should be able to compute the latency distance. The units to use for this are kilometers.

Answer:

The answer is the cost per bit for a hard drive is approximately 0.01 nanodollars per bit. Remember a nanodollar is a billionth of a dollar, so this is a cent of a billionth of a dollar per bit. The latency distance, though, is quite high. It's 2098 kilometers. So that means accessing data from the hard drive is like traveling between Munich and Moscow. This is a pretty big difference between accessing data from the DRAM memory, which is only about 3 meters away, about what you can reach with your arms, which is again very different from accessing data from the CPU, which is the distance between your thumb and your finger.

So when people write programs and they care about performance, you really have to think about memory quite a bit. You have to make sure that the data that you access frequently is stored in these close, fast-to-reach places, not in the hard drive, which is like traveling between Munich and Moscow to get to your data. And no one is storing data in light bulbs.

Programming Quiz: Loops On Lists

The final major concept we're going to introduce in this unit is how to loop through a list. Since loops are collections of things, it's very useful to be able to go through a list and do something with every element of the list. With the things we know so far, we already know enough to do this if we put them together in the right way. You should remember the while loop from the previous unit. And as a reminder, this is the way the while loop works. We have a while followed by a test expression, which is something that evaluates to true or false. And when it evaluates to a true value, that means to execute the block, which can be a block of any number of statements we want. And then at the end of the block we continue, go back to the while loop, try the test again, and we keep executing the block as long as the test evaluates to true.

So for this quiz we're going to see if you can figure out how to use while to go through the elements of a list. We're going to provide most of the code for you. Our goal is to define a procedure called print_all_elements that takes as input a list. We'll use the name p for the list. Our goal is for that procedure to print out every element in the list. I've written most of the code for this for you, but I've left out the test expression for the while. So the code that's written introduces a variable named i, initializes this value to 0, and we're going to use i to index through the elements of the list. For each element we print out that value, and then we increase i + 1. So can you figure out the test expression we need for the while to make print_all_elements work, and what print_all_elements should do is print out every element in the input list p.

Answer:

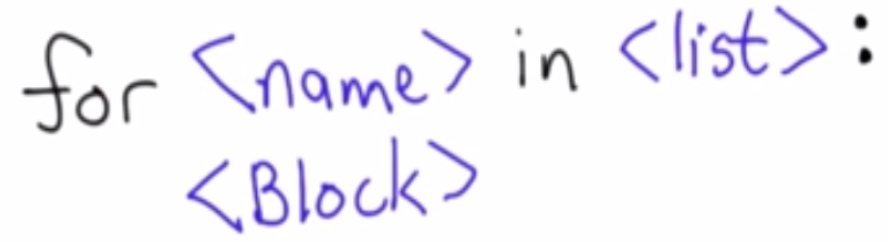
Here's the answer. When the procedure starts, p refers to some list that was passed in. We want the procedure to work on any list, so we shouldn't assume anything about the list that was passed in. It could be any length. And i has the value of 0. That means the value of p index i is this first element of the list p. In the loop body we print out that element, and then we go on to the next element by increasing i by 1. i will now have the value 1. We go to the next element of p. We're going to keep on going until we get through the elements. What we need for the test condition of the while is to figure out when to stop. We learned about the len operator. The len of a list tells us the number of elements in the list. The highest index in the list is not the value of len because the list starts from 0. The highest index is the value of len(p) - 1.

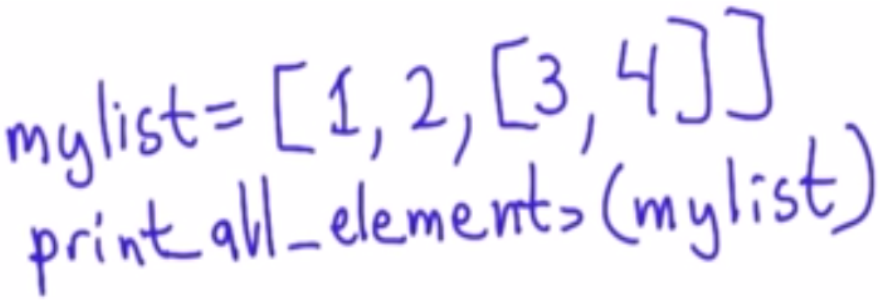
So we want the test condition of the loop to make sure that i has not exceeded that last index. We could write that condition many different ways. One way would be to use i < = len(p) - 1 since we know i is an integer since we created i, initialized it to the value 0, and the only thing we do with i is add 1 to it. This is equivalent to a simpler expression: i < len(p). Either of these would work as the test condition for our loop. We'll use the shorter one since it's simpler.

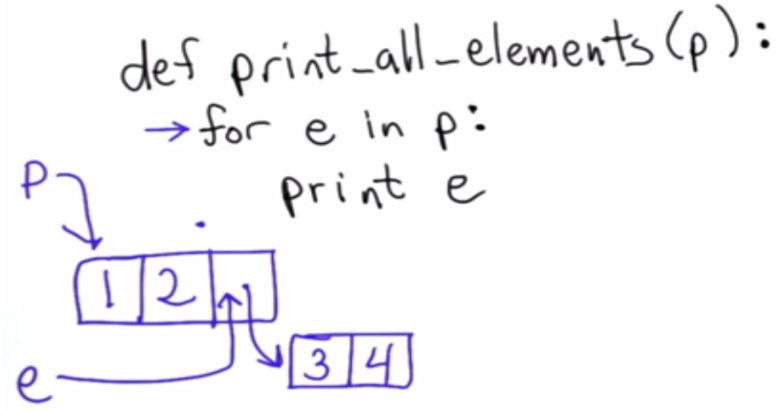
For Loops

So we could write all the code we need to loop through elements of a list just using while. We'd be able to do anything we want. It would just be a little more complicated than we want. Python provides a simpler way to loop through elements of the list, and that's called a for-loop. The structure of a for-loop is like this. We have the keyword for

followed by a name, and this is a new name for a variable that we can introduce, then the keyword in followed by a list, and this is any expression which evaluates to a list followed by a colon. So this is quite similar to what we've seen for the structure of a while loop and an if statement with a block inside the for. What a for-loop like this means is that for each element in the list we're going to assign that element to the name and evaluate the block. And we'll do that in order going through the list. So using for, we can define the procedure print_all_elements using much less code than we needed using while. We have a for statement where we'll introduce the name e as the variable name. We're going through the list p. For each element what we want to do is just print that element. So let's step through what happens when we use the for-loop calling print_all_elements, passing in the list we've defined and stored in the variable mylist. This is a list with 3 elements. The first 2 are numbers, 1 and 2, and the third is the list 3, 4. So when we created mylist, that created a list object that looks like this. It has the 3 elements, numbers in the first 2 and the list 3, 4 in the third one. When we pass that to print_all_elements, the variable p will refer to that object. When we execute the loop, what happens is the variable e is assigned to the first element of p. So initially, we will refer to this value. We execute the body of the loop with that as the value of e, so that will print out the value 1. Then we continue. The next time through the loop, e will refer to the second element of p. We evaluate the body of the loop, printing out that value, which is 2. Then we continue in the loop. The next time, e will refer to the third element of p, which is the list 3, 4. This will print out the list 3, 4. At this point we've gone through all the elements of p and the for-loop is done. Execution would continue here. In this case there's no statement there, so we're done executing the procedure and we return.







Programing Quiz: Sum List

For this quiz your goal is to define a procedure, named sum_list, that takes as its input a list of numbers and produces as its output the sum of all the elements in the input list. So here's an example. If we pass in the list containing the three numbers 1, 7, and 4, the result of the call to sum_list should be 1 + 7 + 4, which is 12. See if you can define sum_list using the for-loop that we've just introduced.

Answer:

Here's one way to define sum_list. It's a procedure that takes in 1 input, and we'll name the input p as the name for our parameter. I tend to use p as the name for a list. There's no really good reason for that. I don't want to use the letter l because l looks too much like a 1. I don't want to use m, n, or o because m and n are used usually to represent numbers and o looks too much like a 0. So the first good letter to use after l is p. Of course you could use any name you want for your parameter name. To define sum_list we're going to introduce a variable to keep track of the sum so far. Let's call that result. Initially, the value of the result of 0. Next we'll use a for-loop to go through the elements. We're going to go through the elements of p.

Each time we execute the block of the for-loop the value of e will be the current element of p, and we'll go through the loop once for each element of p in order. So what we want to do is add that value to result. When we then get to the for-loop, that means we've gone through all the elements and we should return the result. Let's try this in the Python interpreter. We've defined sum_list. Let's try running it. We do get the value 12. What do you think would happen if we run sum_list like this, passing in the empty list? The answer is that we get the value 0, which is what we expect, right? The sum of no elements is 0. And if we look at the code, that's exactly what happens. Initially, we set the result to 0. The for-loop goes through all the elements of p, but if there are no elements of p, it doesn't execute at all. And so we go to the next statement, which returns the result, which has the value 0.

Programming Quiz: Measure Udacity

[Evans] For this quiz, which is an especially udacious one, your goal is to define a procedure, named measure_udacity, that takes as its input a list of strings and outputs a number that is a count of the number of elements in the input list that start with the letter U, where U has to be upper case since lower case Us are not quite udacious enough for us. For example, if we evaluated measure_udacity passing in as input the list containing the strings Dave, Sebastian, and Katy, the result would unfortunately be a 0 since none of those strings start with an upper case U. If we instead passed in the list containing the strings Umika and Umberto, the result would be 2 since we have 2 strings that both

start with the capital letter U. See if you can define the procedure measure_udacity. I can't claim that this will be a really useful procedure, but it will be a good test of whether you understand how to use a for-loop to go through the elements of a list.

Answer:

Let's define our procedure in the Python interpreter. We'll give it the name measure_udacity, and we'll give the parameter the name p. Actually, we should probably give the parameter the name U, and we'll use an upper case U. Not the most standard name for a parameter but appropriate in this case. We'll introduce the variable count. We'll set its initial value to 0. We're going to use count to keep track of the number of strings that we found that satisfy the udacity property. And we'll use the for-loop to go through all of the elements in U. Now we need to decide whether or not to count this element. We can use an if statement to do this. We're going to need to use a comparison to check if the first character of the string is an upper case U. If it is, we want to add 1 to the value of count. If it's not, we do nothing. So we use the equality test to check if the value at position 0 in e, the current element of the list, is an upper case U. If it is, we add 1 to count.

If not, we don't need to do anything, so we don't need an else clause. And now we just need to return the value of count. Let's test our procedure. Our first example, the input strings were Dave, Sebastian, and Katy. We see the result is 0, as expected. We'll try the second example where we have 2 strings that start with the letter U and see that we get the result 2 as the second value printed out. We should also test a string that has a mix of Us and non-Us. Now we're passing in 4 strings. They all start with a U, but only 2 of them start with an upper case U. So the result for the third number printed out should also be a 2. And we see that we get the expected result.

Programming Quiz: Find Element

I hope you're starting to feel fairly confident that you understand lists, you understand the operations we can do on lists, and you understand how to use for and while to loop through elements of a list. For this quiz we're going to try something a fair bit tougher than what you've seen so far. Your goal is to define a procedure, named find_element, that takes 2 inputs. The first input is a list. It can be a list of any type of element, and the second input is a value which can be of any type, and it outputs the index of the first element in the input list that matches the value that's passed in as the second input. One of the things you should think about when you define a procedure is whether there are any special cases that it's not clear what you should do. In this case the question as I've stated it so far doesn't explain what to do if the input list doesn't contain any element that matches the value that's passed in as the second input. We'll change it to make that more clear, and we'll make it behave like the find method that we saw in strings.

So if the element doesn't exist, we'll return -1. Here's a few examples. If you call find_element, passing in as the list the list 1, 2, 3 and passing in as the value to match the number 3, we find the match here, which is position 2, so the output should be 2. If we pass in as the list the list containing the strings alpha and beta and we pass in as the element to find the string gamma, there's no such element in the list, so the output should be -1. There are many different ways you could define find_element using the things that we've seen so far. See if you can find a way to define it that has this behavior.

Answer:

There are many different ways we could solve this problem. First we'll look at a way of solving it using a while loop. We'll define our find_element procedure. We'll use p as the name of the list and t as the name of the target, the element that we want to match. We'll define the loop to go through the elements of p, similarly to the previous while loops we've seen that go through lists. We have our while loop. We've introduced the variable i. We'll use that as the index to go through the loop. The stopping condition for the while loop is when i reaches the len(p), so we want our test for the while loop to be i < len(p). In the block of the while loop we want to check if the current element matches t, so we get the current element using p[i] and then we use == to test if it's equal to t. If we find a match, meaning the 2 values are equal, then we want to return the index that we found. So the result should be the value of i, which is the index where we found the matching element.

The risky thing about using while loops instead of for-loops is it's really easy to forget that you need to increase the index variable. If we just left the loop like this, it would run forever because the value of i would never change unless the first element matched, in which case we would return 0. The loop would just keep going on forever, again checking the first element. So we need to increase the value of i, and that's the end of the while block. The way we described what the find_element procedure should do, if the element was not found, it should return -1. If we get to the end of the while loop without returning, that means we've gone through the while loop for all values of i up to len(p) - 1. We didn't find any element that matches, so we should return -1. So that's one way to define find_element. I'll also show you a way to define it using a for-loop.

The reason it's more natural to start thinking of defining find_element with a while loop than a for-loop is because the value that we want to return from find_element is the index itself. When we use a for-loop with the standard syntax of going through the elements of the loop, we don't keep track of the index. We just see each element in order. So we need to add something else to keep track of the index because that's the value we want to return. So we still need to use a variable to keep track of the index. We'll use i as that variable just like we did in the while loop version. Now instead of a while loop we'll have a for-loop. We don't need to think of the stopping condition here because it will just go through all the elements. Similarly to the body of the while loop, we check if the current element is equal to t. In the for-loop we can get the current element using the variable e. That's what gets assigned each time we go through the loop body to the value of the current element. So our test is

using == to compare e and t. If they match, just like we did in the while version, we should return the result. The result we want to return is the index where we found the match. In the while version that was clear. It made sense because we were looking at element p, index i. In the version with the for-loop we have to be more careful to know where that index is. We're using the variable i to keep track of that index, so we'll return i. i starts at 0.

Each time through the loop we need to increase i so we keep track of the current index as we go through the elements. As with the previous definition, when we had the while loop, if we got to the end without finding it, that meant that the element did not exist in p and we should return -1. We'll do the same thing here.

Programming Quiz: Index

There are many other ways to define find_element. I'm going to show you 1 way that takes advantage of another built-in list operation that we haven't introduced yet but that makes it much easier to write find_element. The operation is called index. The index method is invoked on a list. You pass in a value, and the output in index is the position that that value exists in the list. So the built-in index almost solves find_element exactly the way we want. The difference is when we invoke index on a list that doesn't contain the value we pass in. Index doesn't return -1. What it does is gives us an error. So let's look at what happens in the Python interpreter. I'll define a variable p and give it the value of the list 0, 1, 2. If we invoke index on p, passing in 2, we get as the result 2, which is the position in p where the value 2 occurs. For the second example we'll add more 2s in the list, and we see that it always gives us the first one. So the result returned by p index is the first place in the list where that occurs. Now we'll try an element that doesn't exist in the list. What we see is instead of getting -1 as a result we get an error that the target element that we looked for doesn't appear in the list.

So if we want find_element to have the behavior that's described, we can't quite do it using index directly. There are some other list operations that might be very helpful, though. So to summarize the behavior of the index method, if the value passed in is in the list, it returns the first position where the value is found. This is exactly what we wanted for find_element. The problem is if the value is not found in the list, it produces an error. There's another list operation that will be useful to allow us to use index to find_element, and it's useful for many other things, and that's the in word. We've already seen in being used in the for-loop. Here we use the same word in, but in this context it means something different.

The syntax is to have a value to the left of the in word followed by the list. The syntax is a bit strange compared to the other things we've seen in Python. It doesn't look like a procedure call, but it's very natural in terms of how it corresponds to how we would say this in English. If we say something like, "Is 3 in the list?" we would write that in Python as 3 in p. So let's see how that works in the interpreter. Now I'm printing the result of 3 in p. If I change this to print the result of 2 in p, we get the

value True. So to summarize the behavior of the in operator, if the value is in the list, the output is true. Otherwise the output is false. Similarly, we can use not in. Not in has the opposite meaning of in. If the value is not in the list, the result of value not in list is true. If the value is in the list, then the result of value not in list is false. The value not in list is exactly equivalent to saying not value in list. The only reason to have the not in is it's more natural to read it this way in English than having the not between the value and the in.

So now to check that you understand index and in and not in operations, I want you to try to define the find_element procedure again. It should have the same exact behavior as the ones we've defined before. It should give us the position of the element we're searching for if it exists, and it should give us -1 if it doesn't exist. But this time instead of using a while loop or a for-loop to define it see if you can define it using index.

Answer:

Here's 1 way to define find_element using index. In the case where the element does exist in p, index gives us exactly the result we want. So we can use an if statement, check that t is in p using t in p, and then if it is, the block for the if returns that value. It returns p.index(t). For the else clause the element did not exist in p, so we return -1. Here's another way we might define find_element using index. Instead of checking if t is in p this time, we'll check if t is not in p. If t is not in p, then we know the result is -1. We can return -1 right away. If we reach the next statement, then we know that t is in p and we can return the result of p.index, passing in t. So we've seen 4 different ways to define the find_element method. They all have the same behavior, but we've defined it once with a while loop, once with a for-loop, and 2 different ways using the index method. When we use index, we need to use an if statement as well to make sure that we produce the right result for the case where the element is not in p.

Guest Speaker

DE: This is a long unit, and I know things are definitely getting tougher. If you are having trouble I hope you will take advantage of the course forum, there is lots of great discussions there with other students in the class, as well as the course staff. I want to make sure everyone sticks with the class, so

we asked Sergey Brin to explain why you should do that.

ST: So will you entice students to stick with the class and computers?

SB: I certainly recommend that you continue with the class and go all the way through it to completion to really make sure that you get a deep first cut at computer science and at programming.

Programming Quiz: Union

We'll have 1 more quiz about using lists. It will combine many of the things that we've learned. After this quiz we'll be ready to build our web crawler. For this quiz your goal is to define a procedure, named union, that takes 2 lists as its input. It should modify the value of the first input list so when union returns its value is now the union of the 2 input lists. When we compute a set union, we want to add all the elements in the second list to the first list, except if they already exist. We shouldn't be creating any duplicate elements by adding elements that already exist.

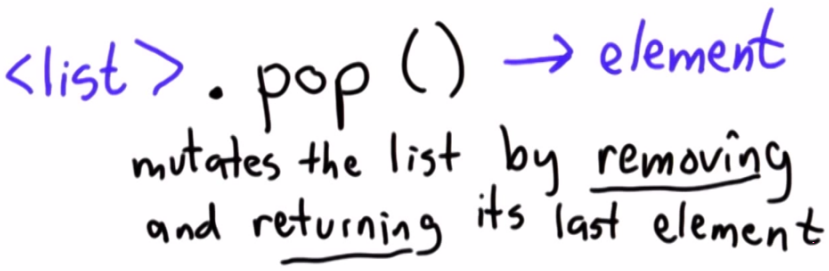
So here are a few examples. If you start with the variable a, referring to the list 1, 2, 3, and the variable b, referring to the list 2, 4, 6, then we call union, passing in a and b. After the call, the value of a should be the list 1, 2, 3, 4, 6. We've added the elements of b to a, except we skipped the first element, the value 2, since 2 already exists in a. I should mention that union should not modify the value of b. After the call to union (a, b), the value that b refers to should still be the list 2, 4, 6. So see if you can define the union procedure as described here.

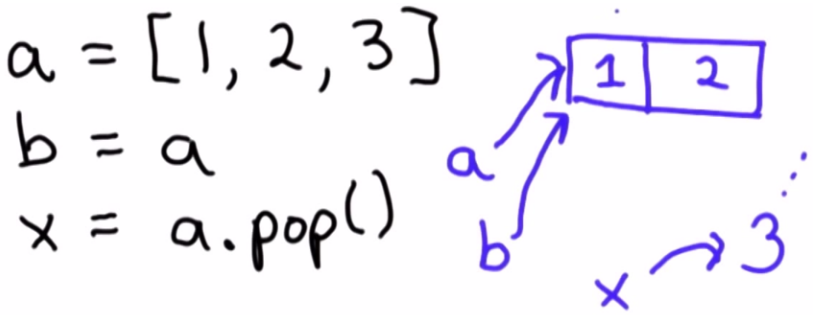
Answer:

Let's think about how to define union. We need to make a procedure that takes 2 lists as its inputs. We'll call the lists p and q. To define union we want to modify p by adding elements from q. We should remember that append operation does that. It adds a new element to a list. So we're going to use append to add elements from q to p. To define union we need to go through all the elements in q. Check for each element whether it already exists in p. If it does, we do nothing. If it doesn't, then we add it to p. So we'll use a for-loop. We need to make sure that we loop through the elements of q. Those are the ones that we want to add to p. For each element we need to test whether or not it's already in p, so we check to make sure the element is not already in p. If this is true, that means e is not in p, and we want to add it. To add e to p, we use append.

Pop

We're going to introduce one more list operation that will be very useful in building our web crawler and that's the Pop operation. Pop works like this. We have a list. We have .pop. This is similar to the way we use the find method. There are no inputs to pop, but we still need the parenthesis to show that we're calling a procedure and what it outputs is an element of the list. What pop does is it mutates the list. It removes the final element of the list. It also returns the value of that element. So, here's an example. If we started by initializing the variable a to be the list one, two, three. And then, we'll make b an alias for a, so here's what we have after those two statements. We have created a new list with elements one, two, three and a refers to that list, after the assign a to b statement, b also refers to that list, then we do x assign a pop to x. So the value of x will be three, that's what the last element was. That's what we get returned by pop. But it also mutates the value of the list. It removes that element. So now a refers to the list one, two. Since b was an a list for a, it also changed the value of b. b is now the list one, two.





Quiz: Pop Quiz

Now it's time for our pop quiz to make sure everyone understands pop. For this quiz, assume that the name p refers to a list with at least two elements. Your goal is to determine which of the code fragments does not change the final value of p. And so that means no matter what value p starts with, as long as it has at least two elements after executing the code. And I'll show you the choices for the code next. The value of the p is the same as what it started with. So here are the choices. Check the box for each code fragment where the value of p at the end of the code fragment is the same as the value of p beforehand, assuming that initially p has at least two elements.

[x] p.append (3)
    p.pop ( )
[x] x = p.pop ( )

```
    p.append (x)
[ ] x = p.pop ( )
    y = p.pop ( )
    p.append (x)
    p.append (y)
[x] x = p.pop ( )
    y = p.pop ( )
    p.append (y)
    p.append (x)
```

Answer:

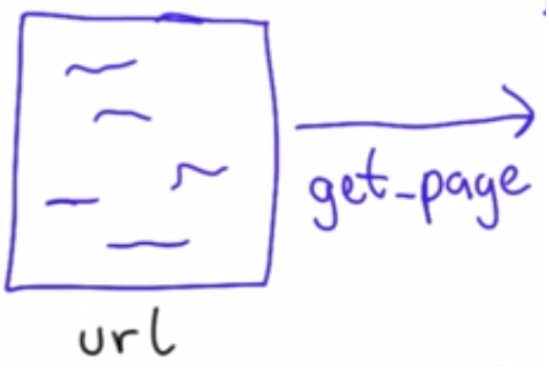
So the answer is all of the code fragments except for the third one, except for this one, leave the value P unchanged. So let's work through why that is. We're not going to assume anything about P. It's, list with at least two elements. So I'll draw P. We can't make any assumptions about what the elements are. We'll call them Alpha, Beta and Gamma. Actually, it looks like alpha, beta, and alpha. That's okay. So let's look at the first option. Here's what happens. We append a 3 to P. That adds a new element at the end of P. And then we pop. That removes it. We don't do anything with the result returned by pop, but that's okay. The value of p is unchanged. For this choice, we start by popping. That will remove an element from p, so now x will refer to whatever this element was. So x refers to alpha in this case, and p is mutated by mod, removing that element. So at this stage, the value of p has changed, but we're not done. We still have the append to do.

The append appends to p, adding onto the end of p the value that x refers to, which is the alpha value that was in that position to begin with. So at the end of these two steps, the value of p is also unchanged. Let's look at the third option and see why that does affect the value of p. So the first statement, we assign to x the result of p.pop(). That will remove the last element from p and make x refer to it's value. So now whatever value was at ten of p the name x refers to. Then we do another pop. So now after this pop, y will refer to the value here, which is beta. We remove that value from p. So at this state p has lost two elements, but we've stored them in the variable x and y. Then we do the append, and we're doing the appends in the same order we did the pops.

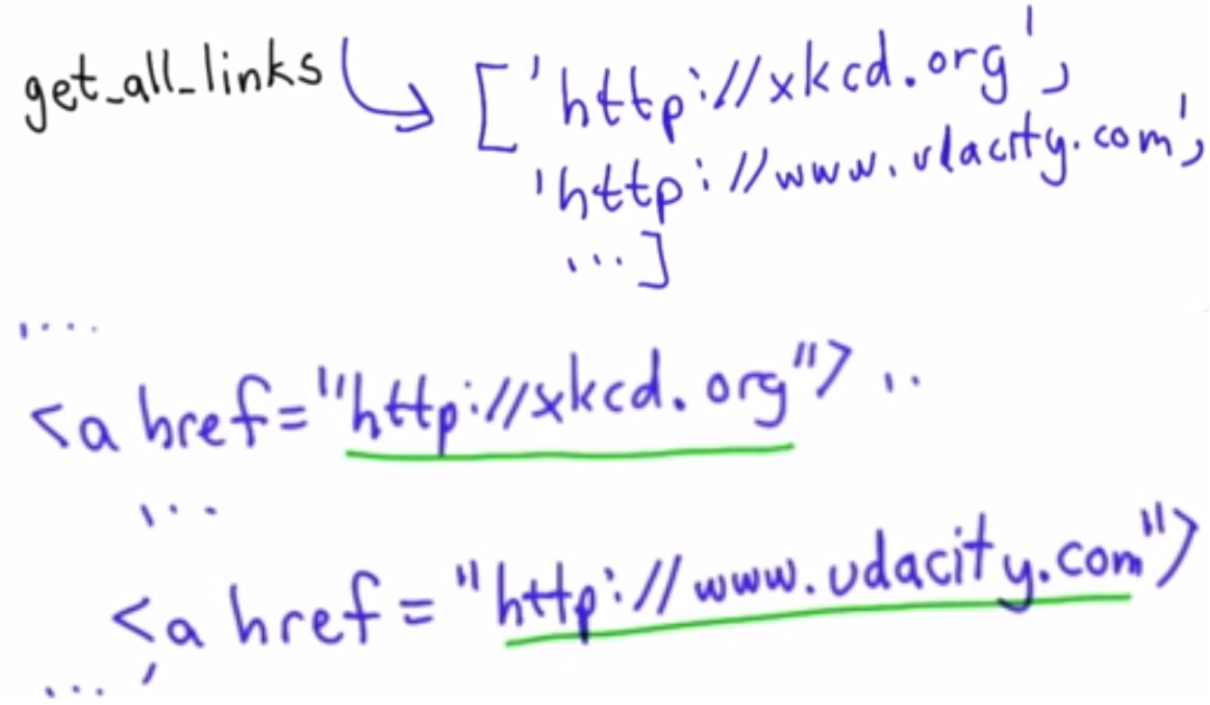
So we're appending x first. The value of x is alpha, so this will Add the alpha to p. Note that that's not what we started with. We started with the beta there. Then we do the next append. That will add the value of y, which is beta to p. If we pop elements of p, and we add them in the same order that we pop them, this will actually reverse the order of the elements, since each time we append, we add to the end of the list. So for the final choice, we're again popping two elements. This time we're appending in the opposite order, that means when we do their appends so the value when we popped x, that got the value alpha. When we popped y, y was the value beta. P lost these two elements as the result of the two pops. Now we're here. We do the appends. And this time, we're appending y first. Well, that's

the beta, that's the last one we popped, the first one we append. This gets the elements in the order they were before. And finally, we append x, which is the alpha. This restores the value of p to what we started with.

Collecting Links



So now we're ready to get back to the problem of extracting a list that contains all the links in a webpage. This is the first step towards our crawler, which will crawl a set of web pages, finding all of the links that can be found from a seed page. We're going to start with a web page. We have the url of some seed page. We use the get_page procedure to get a string which is the text of that page. That's got lots of stuff in it that we don't care about, but it also includes some hyperlinks, and inside the hyperlinks are the url's to that page links too. Our goal is to define a procedure--we'll call it get_all_links-- that takes as input, a string representing the text on a webpage, and produces as output, a list containing all of the url's that are linked to by that page.

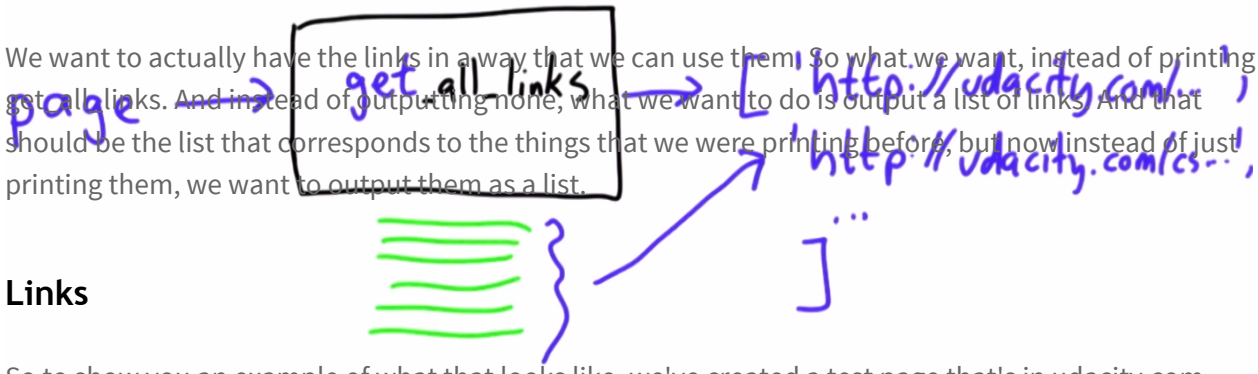


Get All Links

So let's recap the code we have at the end of unit two. So we defined a procedure, get_next_target, that would take a page, search for the first linked target on that page, return that as the value of URL.

That would be the first output, and also return the position where the end of the quote is so we know how to continue. And then we define the procedure print all_links that keeps going as long as we can. As long as there are more URLs on the
page. It will find the next target. Store these in the variables URL and endpos to keep track of the location end of string. If there is a URL, what we did was just print it out and then we update with the page to keep going. What we want to do to change this is instead of just printing out each URL as we find it, we want to collect the URLs. We want to have a way to use the URLs so we can use them to keep crawling to find new pages. The structure we've been learning about this unit is the way to do that. What we want to do is keep all the URLs in a list. At the end of this procedure, instead of printing the links as we go, we want to have a list of all the links that we found. So this is what the current print_all_links procedure does. It takes the page as its input and its output is nothing. It doesn't return anything. All it does is do some work, prints out all these links. But we can't actually use them at the end, because it doesn't return anything. So what we want to do is change this. Instead of
p rint_all_links, what we want is to

We want to actually have the links in a way that we can use them. So what we want, instead of printing all links is to ac
get all links. And instead of outputting none, what we want to do is output a list of links. And that should be the list that corresponds to the things that we were printing before, but now instead of just printing them, we want to output them as a list.



Links

So to show you an example of what that looks like, we've created a test page that's in udacity.com cs101 index.html and when we run get all links we'll see all the links on that page. Let me show you what the page is first. This is our test page for learning to crawl. So that's the URL that we're using to get page and get all links from that pag
goes to crawling.html, the second one
We'll look soon at the structure of these
own in the web browser. For now, what

```
2  def get_next_target(page):
3      start_link = page.find('<a href=')
4      if start_link == -1:
5          return None, 0
6      start_quote = page.find('"', start_link)
7      end_quote = page.find('"', start_quote + 1)
8      url = page[start_quote + 1:end_quote]
9      return url, end_quote
10
11 def print_all_links(page):
12     while True:
13         url, endpos = get_next_target(page)
14         if url:
15             print url
```

So when we assign links for the result to get all links. We can print the value of links that we get back. And when we run that, we see a list containing the three links on the page. And because that is a list, well, we can do useful things with it. We could extract the value at position zero, and get just the first link on the page. So think on your own, if you could define get all links. If you're ambitious, you'll try to define this all by yourself. We're going to step through how to do this with a few quizzes because I think it's quite a challenging thing to do all by yourself. But think about that by yourself first, and then we'll step through it with a series of quizzes.

Programming Quiz: Starting Get All Links

So here's the code, that we had at the end of unit two. Our goal now is to modify that code, instead of printing all links, to get all links. The good news is, the get_next_target we don't have to change at all. We still want to find the next URL in the page. We're going to need to make a few changes to print all links to make get all links. But not too many. The first change we need to make is we need to introduce a variable to keep track of the results. So I'll call that variable links. I'll leave it as a quiz for you to figure out what the initial value of links should be. Remember, our goal for get_all_links is that at the end of get_all_links, it will return a list containing all of the links that we found on this page.

Answer:

So the answer is we want links to start out as the empty list. And we get the empty list with the two square brackets.

Programming Quiz: Updating Links

The next change we want to make is we don't want to print the URL. We want to do something else with it. For your next quiz, your goal is to figure out the code that we need here. And remember, what we want to do is to maintain in the variable links, a list of all the URLs that were found on the input page.

Answer:

So the answer is we want to append the URL to links. So we can do that by using links.append, passing it as the input, the URL that we found as the next target. So this will add that value to the list that links refers to, keeping track of all the URLs that we found on the page.

Programming Quiz: Finishing Get All Links

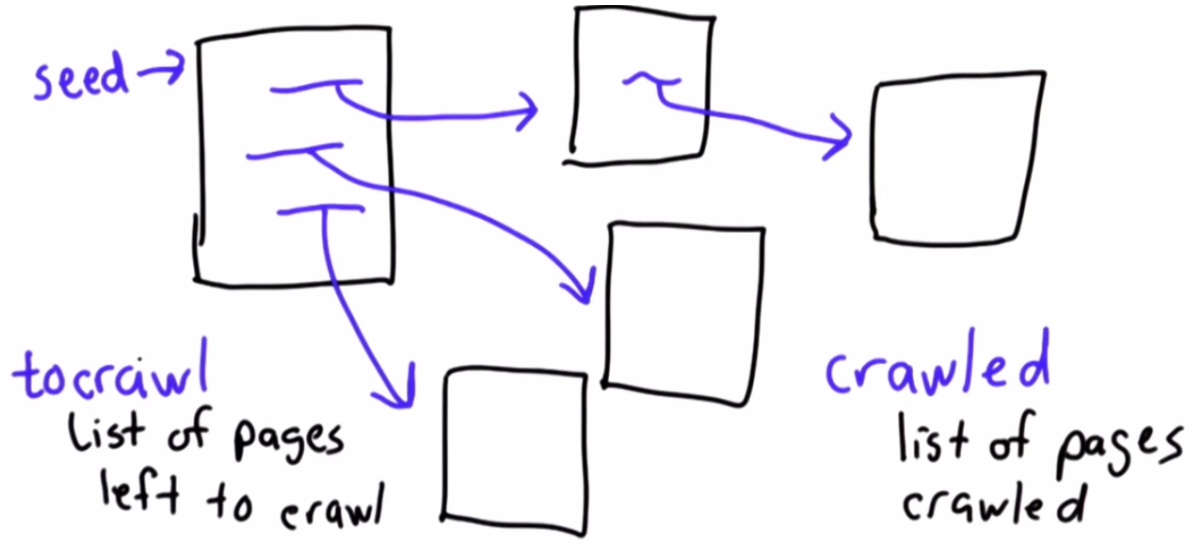
The last thing we need to finish to get all links is to produce the output. So I'll leave it up to as a quiz to figure out what we need here to do that.

Answer:

The answer is we need a return statement. And we notice it's indented at the outer level. And what we want to do is return the value of links, which is the list of all the links that were found on the page.

Finishing The Web Crawler

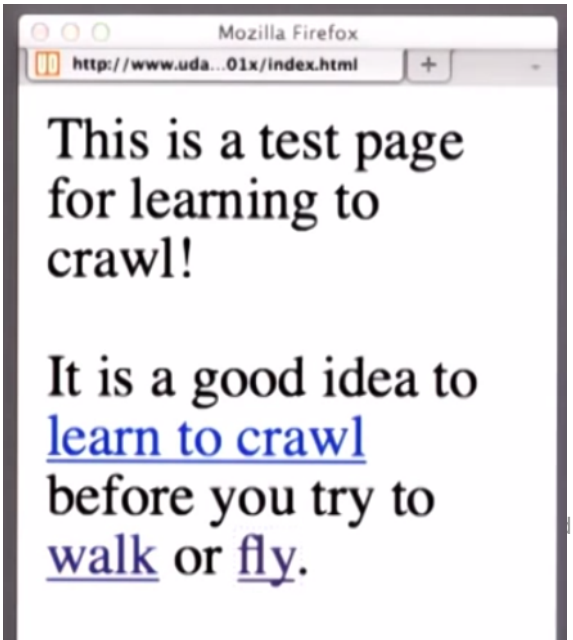
So now we're ready to finish the Web Crawler. And remember what we want the Web Crawler to do. So, we have a seed page, and we're assuming we know some seed page to start with. And the seed page has some links on it. We want to be able to find those links, and we know how to do that now. We're going to get them to list, and then we're going to follow those links. So we'll follow the links. To new pages, and those new pages might also have links, and we want to follow those links. So in order to do this, we need to think about two things. We need to keep track of all the pages that we're waiting to crawl, and we'll introduce a variable, tocrawl, to do that. And what tocrawl will be is a list of pages left to crawl. So, initially it'll just be the seed page. Once we collect the links on the seed page well it will include those links as well. Once we finish crawling a page, we don't want to keep it in tocrawl anymore. And as we find new pages to crawl, they'll be added to the tocrawl list. The other variable we want is to keep track of all the pages that we've crawled. At the end of our crawl, this is the result. We want to know all the pages that we found.



That will be stored in the variable we'll call crawled. So let's walk through an example of how this should work on the sample site. So I'll make the seed page, www.udacity.com/cs101x/index.html, that's this page here. That means when we start to crawl, we want tocrawl to be this index page. And I'm going to stop writing out the full URLs, just writing out the final part. because all the pages we crawl will be on our test site. So tocrawl will be the list containing just one element. The index.html page. We haven't crawled anything yet, we're just getting started, so crawled will start out as the empty list. The next thing we're going to do is crawl this page, so we'll get all the links on this page. That means we've crawled the index page, so that will now be added to crawled. But when we looked at the links on the index page, we found three new links on that page. We found a link here, which goes to crawling.html. We found the link here, which goes to walking.html. And we found the link here that goes to flying.html. So the new value of tocrawl, after crawling this page, will have those three links in it. The next thing we want to do is take one of those links and crawl it. The order actually matters a lot in terms of getting a good crawl. Let's assume for now that we'll do just do the last one first. So, we'll do the link fly. That links to the flying page. Here is the page there. So, we are going to crawl the page flying.html. This page doesn't have any links. If you're not sure why squeamish ossifrage is the magic words, I would encourage you to DuckDuckGo or Google that. And now we've finished crawling flying, so that's going to be added to the crawled list, which already had index.html, we don't lose that. We're going to add the new one, which is flying, to that list. And we finished crawling it so we don't want to crawl it again. Let's remove it from the tocrawl list. Now, after we're crawled flying, we have two more links left in our tocrawl list. We have two links that we've crawled. So let's try another link. Let's suppose we follow the crawling.html link. And we follow crawling. We get to this page. So, to follow crawling, we're going to follow the same algorithm we did with flying. Alright, so that removes this link from the tocrawl list, adds it to the crawl list. So, we're done crawling, crawling. And now we want to add to our tocrawl lists all the links that we find on this page. Well, we found the link, kicking, which goes to the page, kicking.html So we're going to add that to our list of pages to crawl, and now we keep going. And we're going to keep going. We'll follow kicking. We find that kicking does not have any links. So that would add kicking to the crawled

list and remove it from the to crawl lists. And we're going to keep going until we have no more pages to crawl. So let m

more formally and then I'll ask you a question about it.



Quiz: Crawling Process

So I'm going to describe that process, and I'm going to write it out in a, fairly precise way, but not as actual python code. Because it will end up being your job to finish the python code to do this. But I want to describe it precisely enough so we can ask some questions about it. So we're going to start with some seed page and to crawl will just be that page. The list containing just the seed page, and crawled will be empty. And we're going to keep going as long as there are more pages to crawl. And for each step we're going to pick one of the pages, we'll add that page to crawled to keep track of the fact that we've crawled it. Then we'll find all the link targets on this page, and add those to tocrawl, and we'll keep going as long as there are more pages in tocrawl. And then when we're done we'll return crawled. So that's the basic process that I've walked through. So for this quiz. I want to see that you understand the process that we're using to crawl the web, well enough to understand what will happen if we do this starting from the seed page on the example test site. And you can try exploring that site yourself. So the choices are, it will return a list of all the URL's that can be reached starting from the seed page. The second choice it will return a list of some of the URL's but not all of them that are reachable from the seed page. And the third choice, it will never return.

[ ] It will return a list of all the urls reachable from the seed page
[ ] It will return a list of some of the urls reachable from the seed page
[x] It will never return

Answer:

So the answer, is that it will never return. And the reason for this is the stopping test for the while loop is when there are any more pages to crawl, so we're going to keep doing this as long as there are pages in tocrawl. So in order to finish, we need to know that the value of tocrawl eventually becomes empty and if we look at the test site. If you follow the walking link, which the crawler will do, you get to this page that has a link to crawling that goes back to the index. That will keep going, and you're going to follow the walking link again, follow the crawling link again, following the walk, walking link, following the crawling link back to the index. And this will continue forever.

The crawler will never finish because it will always find a link to crawl. And the real web is full of circular links like this. There are lots of pages that link to each other. There are also pages that link back to themselves. So to avoid this, we need to do something smarter. We need to make sure that we don't crawl pages that we've already done. So we're going to have to be a little more careful about this step. We need to add a test to see if we already crawled this page. If we did, we don't do anything. If we didn't already crawl it, well, then, we need to add it to crawl, add all the links in that page to tocrawl and keep going.

Programming Quiz: Crawl Web

Now we're ready to write the code for crawling the web. So our goal is to define a procedure, we'll call it crawl_web, that takes as input a seed page url. So, that's the url that identifies our seed page, and outputs a list of all the urls that can be reached by following links starting from the seed page. So, if you're really ambitious you should try to do this yourself without anymore help. That's going to be a pretty tough challenge. So we're also going to step through one way to do this as a series of quizzes. But you should feel free at any point, when you feel confident that you can do it yourself, to try to finish for yourself, rather than following the step by step quizzes that I'll show you. So we will start defining our procedure crawl web, and we are going to introduce two variables. The two crawl variable that keeps track of the pages that we need to crawl, and the crawl variable that is a list of pages that we have already crawled. For the first step, your goal is to figure out, how to initialize these variables. Which of the first value, to crawl and crawled be?

Answer:

So the answer is we should start to crawl with a list containing just the seed page. And we should start crawled as the empty list. We have not yet crawled any pages.

Programming Quiz: Crawl Web Loop

The next step is to write the loop that's going to do the crawling. And we said the process we want to follow is to keep going as long as there are more pages to crawl. We can do that with a while loop, and we can use tocrawl like this in our test condition. If a list is empty that's interpreted as false. If the list is not empty, that would be interpreted as true. So this means the same thing as testing if the length of the list is zero, it's a cleaner way to write this by just doing while tocrawl. Inside the loop, well, we want to pick a page to crawl. We'll store that in the variable page. And I'll leave it as a quiz for you to figure out a way to pick the page to crawl. There's certainly lots of different ways to do this. If you're clever and can think about using all the things that we learned about, you'll be able to have one line that both initializes page to the next page we want to crawl, and removes that page from the tocrawl list.

Answer:

So the best way to answer this is to use pop. Pop's the only thing we've seen that actually removes elements from a list and it also has the property that it returns that element. If we do tocrawl.pop() that'll get us the last element in the tocrawl list. Remove that element from the list tocrawl. And assign that to the variable page. One important point to note is because we're getting the last element first. What we are implementing is what's called a depth-first search and that means that

as we crawl webpages, If we had three links on the first page. Well, what's going to happen is we are going to follow that last link. We're going to add the links that we find there to the page. Before getting to these links, well, we're going to follow this link, right? That's going to be the last link that we added. We're going to follow that, we're going to get to that page, and we're going to follow the last link on that page. So, this is why it's a depth-first search.

We don't get to look at the second link on the first page until we've followed all the links that we can reach from the last link on the first page. If our goal was to get a good corpus of the web quickly, doing a depth-first search would probably not be the best way to do it. And some of the questions at the end of your homework, ask you to figure out ways to change the search order, that will give us a better way of capturing content on the Web. For now, we're going to be happy, and with the test site, it's enough to do a depth-first search. If we can complete our search, no matter what order we follow, we'll always find the same set of pages. If we aren't able to complete the search, and with a real web crawler, there's far too many pages to wait until we find them all to return a result, then the order that we do the pages matters a lot.

Programming Quiz: Crawl If

So for the next step, we knew to think about this problem rent into web cycles. We don't want to crawl pages that we have already crawled. So what we need is someway of testing whether the page was crawled. If it was crawled, we don't want to crawl it again. If it hasn't been crawled, then we want to do something. So the way to make a decision like that is to use if. So we need a test condition for the if so that will only do the stuff that we do to crawl a page if we have not already crawled that page. So see if you can figure out what the test condition for the if should be.

Answer:

So the answer, is we only should crawl the page if we have not crawled it already. And the crawled variable keeps track of the page that we've crawled already. So what we want to do is test whether the page is not in crawled. If page is not in crawled, then we should crawl it. If it isn't crawled, well, we just keep going. We're going to do nothing else. For this iteration through the while loop, we're going to keep going through the while loop and check the next page. We're not done right away, if this one was already crawled, but we don't want to crawl it.

Programming Quiz: Finishing Crawl Web

So now, we're ready to finish the heart of our crawler. Let me put the last statement in, so you know there's nothing else missing and you'll be able to test this. And the last thing we want to do is return the result in crawled. When we finished the while loop, we're ready to return crawled, which is the

list of pages we found. What we have left to do is to figure out what we do to crawl each page. This is going to be a pretty tough quiz, I think you'll need at least two lines of code. If you think about using all the procedures that you've learned about and the ones that you've defined in earlier quizzes, you shouldn't need more than two lines. And the two things that you need to do are update the value of to crawl to reflect all the new links that are found on page And update the value of crawled, to keep track of the pages that have been crawled. See if you can figure out how to finish the crawl web procedure.

Answer:

So the first thing we'll do, is add all the links that we find on the page that we're crawling, that's the value of page, to to crawl. The best way to do that is to use the union procedure, that we defined in an earlier quiz this unit, that will avoid having duplication in to crawl. If you didn't use union, you could still get the stork. It's okay if there's duplication because we're doing this test to not crawl the same page twice. So I'll do it using union. We're going to union into to crawl. The results of finding all the links on the page that we found, and we need to use get page passing in page to get the actual contents of that page. And the get all links procedure that we defined earlier returns a list of all the links on that page. The next thing we need to do, is to keep track of the pages that we've crawled, we can do that by using append. So that adds this page to the list of pages we've crawled, and now we're done. We've got a working web crawler, for any seed page, it will find all the pages that can be reached from that page, and return them in a list.

Conclusion

DE: So, congratulations. You've built a web crawler. You've learned to crawl. We're not quite at flying yet, but this is a lot that you've learned in just three units. And by learning about lists, we can build any other structure we want starting from lists. And we'll see lots of other things that we do in the next unit, as well as the units that follow that. That use lists. We haven't quite finished building a search engine. We have a way of getting a great start towards our purpose, finding all these pages to crawl. What we need in Unit Four is the way to get the content from those pages, by starting from our seed page, and following the crawl web procedure that you wrote. And then we'll use that to build an index, so we can respond to queries when someone's doing a search for a particular term. The other thing we'll do in Unit Four is, we'll learn more about how the Internet and the web work. I'm here with Anna Patterson who's built a bunch of search engines, including building the world's largest index of the web. And now works for Google. And we've just finished building a very simple web crawler. I want

Anna to tell us something about how things are different if you want to build a web crawler that works with the scale that Google has to deal with.

AP: So there's three main issues with scaling up a web crawler. One is the normal politeness that you need on the Web. The second one is how you get a bunch of machines involved in crawling, not just one. And the last issue is how to consume a lot of bandwidth so that you keep the expensive resource busy. While still being polite. So, on politeness, there's a line in the robots.txt that each domain tells you how often you can crawl that domain. There's a number of problems with that, which is that multiple domains can be hosted with a hosted service, and they can be hosted even on one machine. So even though you're being polite to one domain, you're really actually really hurting a hosting service, or you're hurting a machine. So you have to make plans in order to not hit one machine or one domain too hard. The next thing is, that if you crawl on just one machine. It, the state is very good, and it's really easy to keep the state of the crawl on the one machine. But then it's hard to make a very big search engine, because you get as much as one machine will crawl. So of course in practice, you crawl on thousands of machines. Now, if you're going to obey politeness, that means that each of your thousand machines, needs to tell the other ones, what it has crawled and what it's about to crawl and what it, going to crawl in 20 minutes, right? All of this communication overhead can actually slow down the crawler, which actually hurts your aim of having a big search engine. The last part is that you want to max out your bandwidth. So one thing that people do instead of trying to lessen the communication, is they pre-process the corpus that they already have. So when you've crawled a set of pages you can process them and extract out all the links that they point to. You can then normalize those links so that Yahoo.com and dubdubdub.yahoo.com wind up being the same string, and now you can farm those strings out to your thousand machines, and now they don't have to communicate because they know they're not going to hit the same domain. However, they could accidentally hit the same hosting service and the same machine, but you can take care of that ahead of time as well.

DE: Thanks a lot Anna. I hope those of you who think about unleashing your web crawler on the world, pay careful attention to what she said about politeness.

AP: Thank you for having me, and students, good luck on your search engines.

DE: So I hope we'll see you back soon. You'll have a homework for Unit three that's going to be quite challenging, and I look forward to seeing all of your answers to it.