Intro to Computer Science

Lesson 05 Notes

How Programs Run

Introduction

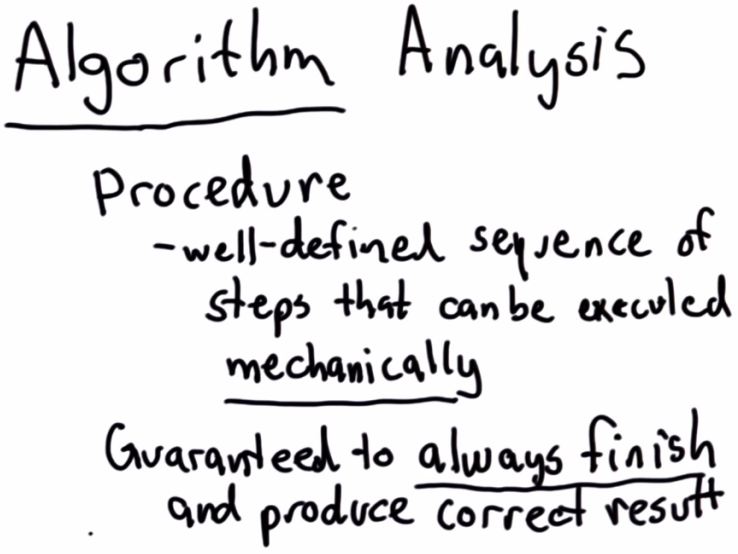

DE: At the end of Unit 4, we built a search index that could respond to queries and would do this by going through each entry one at a time, checking if the keyword matched the word we were looking for, and then responding with a result for that. I wanted to ask Gabriel Wienberg, the creator of DuckDuckGo, how well that approach will work if we have a large index with lots of queries.

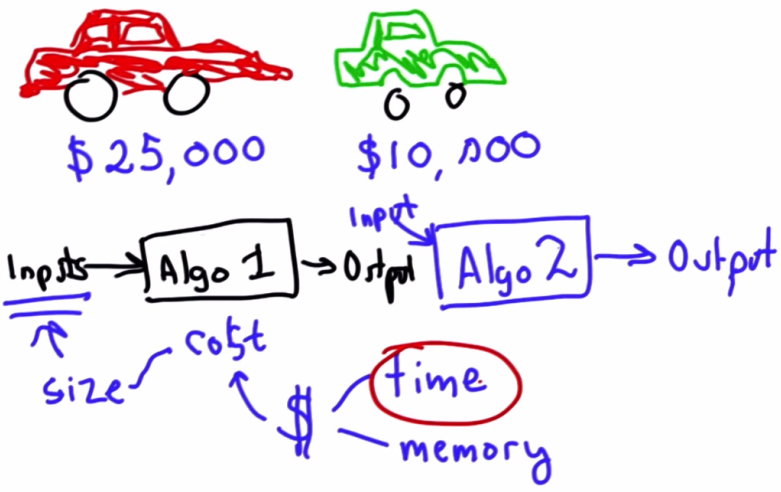
GW: You'll find that with a large index with lots of queries, it will be too slow. A typical search engine should respond in under a second and often times much faster. But with that algorithm, having to go through each link like that one after the other, you could be on the order of seconds or even longer. So what we're going to learn in this unit is how to make this much faster.

Making Things Fast

So welcome to Unit Five. The main topic for this unit is trying to understand the cost of running programs. So far, we haven't really worried about this. We've been very happy to write code. If we get the correct result, that's a great thing. But once we start making programs bigger, worrying about programs that do more things, running on larger inputs, we have to start thinking about the cost of running our programs. And this question of what it costs to evaluate an execution is a very important problem in computer science. In some sense, it's one of the most fundamental problems. Many people spend their whole careers working on this. And it's a problem called Algorithm Analysis. I

haven't yet explained what an algorithm is. But you've actually written many of them already. So an algorithm is a procedure that always finishes. A procedure is just a well-defined sequence of steps. It has to be defined precisely enough that it could be executed mechanically. So to be a procedure, it has to be something that can be executed without any thought. And we're mostly interested in procedures that can be executed by computers. But the important part of what makes it a procedure is that the steps are very precisely defined and don't require any thought to execute. To be an algorithm, it has to always finish, and we've pointed out already that this is a very tough problem to figure out whether a program will finish. In general, it's not possible to answer that question, but for many specific programs it is, and in order for a program to be an algorithm, we have to know that it always finishes, and it always produces a correct result. So once we have an algorithm, well, we know we have a well-defined sequence of steps, so we can reason about what will happen on any i always produce the correct result. So, then we can think algorithm on some input. So how should we think about cost is quite different from how most people think about about cost, well, if you've got a specific object, let's say, y car. It doesn't quite look like a cool car, and it's got a cost $25,000 car, and you have some other car. So you have o costs you get that car. You could another car that would costs $10,000. And when we think about the cost of thing and they have specific costs. And we know that the red ca $10,000; the red car costs more than the green car. We ju about the cost of algorithms, we don't have a specific exe to understand is how the cost depends on input. So we m Let's say this is algorithm one, and we have a second algo of these are algorithms that take inputs and produce output. And if we want to decide which algorithm is better, well, algorithm the way we do for the cars. And we can say the red car costs more than the green car. The cost depends on the actual input that we run the algorithm on, so it might be the case for some inputs algorithm one is faster, and for other inputs, algorithm two is faster, and I should label this algorithm two. So what we need to understand is how the cost of executing the algorithm depends on the input. We don't want to do that for every specific input, right, if we had to do that for every input, well, we might as well just run it on the input and see what it costs. What we want to do is be able to predict this without actually having to run on on every input. Normally, th





ed.

The main thing that's going to
matter about the input is the size
of the input. That's not always
the case, and we'll see examples
where other properties of the
input matter, but the primary way that we talk about cost in computer Science is based on the size of the input. As the size of inc, input increases, how does the cost to evaluate the procedure increase? So cost ultimately always comes down to money. And when it comes down to money, well, what are the things that cost money when we execute algorithms? And the main things that costs money are the time it takes to finish. If we get an answer more quickly, well, we've spent less time on it, and we can also rent computers by the time to execute. There are lots of cloud computing services now that will give you a processor of a certain power for a certain amount of time, for a certain number of cents per hour. So time really is money. So we don't need to turn our cost estimates into money because we don't necessarily know how much our computing cost will be. But if we can understand the time it takes to execute, that will give us a good sense of the cost. The other main cost is often memory. So, if we know that we need a certain amount of memory to be able to execute our algorithms, well, that tells us something about the size of computer we need and how expensive that's going to be. So, we don't usually talk about cost in terms of dollars when we analyze a logarithms, we're talking about cost in terms of time and memory. But those things in real implementations end up being cost in terms of dollars. So we're mostly going to focus on measuring time, and time is usually the most important cost of running an algorithm. Memory is often another consideration.

Quiz: Measuring Speed

So for this quiz, I want you to think about why we're so focused on how time scales with the size of the input, rather than the absolute time it takes for a particular execution to run. Check all the answers that are correct. So the first choice is, we want to predict how long it will take for a program to execute before we actually run the program. The second choice is, we want to know how the time will change as computers get faster. And through the entire history of computing it's been the case that the computer that you can buy for the same amount of money a year from now is faster than the computer that you can get for that price today. So the third choice, we want to understand fundamental properties of our algorithms, not things that are specific to a particular input or machine. And the fourth choice is, we want abstract answers to make sure they can never be wrong.

[x] We want to predict how long it will take for a program to execute before running it
[x] We want to know how the time will change as computers get faster
[x] We want to understand fundamental properties of our algorithms, not things specific to a particular input or machine
[ ] We want abstract answers, so they can never be wrong.

Answer:

So the answer is the first three are all good reasons to focus on how time scales with input size, rather than absolute time. So the first reason is if we have an understanding of how the time depends on input size, well then, we can predict how long it will take before we actually execute a program. If we have to run the program to figure out how long it takes, well that's not going to be very useful because we have actually finished it. We have got the result we want. If we only learn how long it takes on that particular input, we haven't learned anything useful to figure out, what it will cost to run it on some other input. So we want to be able to make predictions, by understanding how the running time depends on the actual size of the input. The second reason is also true. By understanding how time scales with input size, we get a better idea of how the cost will change over time. Computers keep getting cheaper and faster.

This was observed by Gordon Moore in 1965. And turned into the notion that we sometimes call Moore's Law today. It's not a law in the sense of a physical law, but it's a law in the sense that the history of computing has followed this trend, where the amount of computing power you get for the same cost, approximately doubles every 18 months. So what you can get for $1,000 today if half what you'll be able to get for $1,000 a year and a half from now. That's a pretty nice property to have, but it means that understanding the cost of something today doesn't tell us very much. What we really want to understand is the cost in a more fundamental way. The third reason is also true. That by understanding how time scales with input size, we get a much better fundamental understanding of our algorithms, than if we just had some absolute time measurements for a few different inputs. The fourth answer is not correct. Abstract answers can be just as wrong as concrete answers. But having good abstract answers will allow us to understand things much more deeply, than a few specific concrete answers will.

Stopwatch

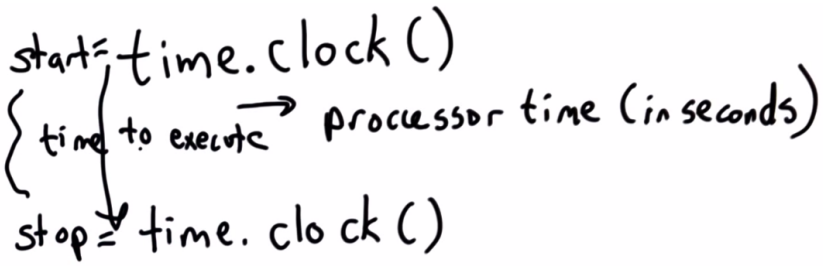
So we're going to try a few things in the Python interpreter to get a sense of how long things take and I've written in a procedure here that times the execution of a piece of code. Right, we could just do the timing with a stopwatch and then we'd have to run really long things to be able to get a reasonably accurate time but Python has a built in procedure we can use to measure time, and that's the time clock procedure. So we could try just using a stopwatch and if we ran programs that took long enough to run, this would give us a reasonable idea of how long they took. It's going to be a lot more accurate to use this built in procedure, which is provided by the time library that evaluates to the number of seconds. So the value of time.clock will give us
the current processor time in seconds. This starting point is fairly arbitrary there but the important thing is if we call it twice, and we start the time here, and we stop the time at the second call, we store those in the variables, start and stop, that's going to give us the amount of time it took to execute this code. So I've rendered a procedure that starts by initializing the variable start to the current clock

time then run some code, and I'll talk more about what it's doing there in a second, and then it computes the time bet
start time, and gives us the run time, and it returns both the results of the code and the time it took to run. What's happening here is actually quite exciting. So we're using Eval. Eval allows us to evaluate any string as if it were a python expression, so we're passing in a string here but when we're passing the string in to eval, it's running it as code. So we can pass at any python expression. We're going to start and end the timer and in between, we're going to evaluate the code and get the result of evaluating that code. So I've defined this procedure time execution that will give us the time it takes to evaluate any Python expression. And now

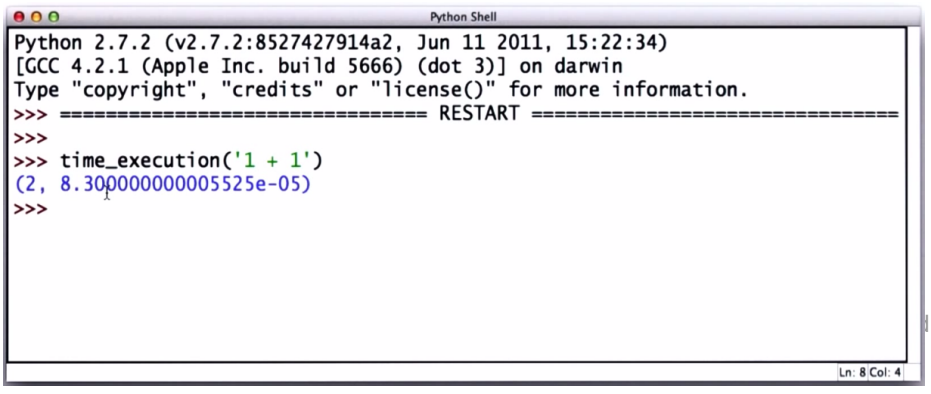
Run

```
1  import time
2
3
4  def time_execution(code):
5      start = time.clock()
6      result = eval(code)
7      run_time = time.clock() - start
8      return result, run_time
9
10 def spin_loop(n):
11     i = 0
12     while i < n:
13         i = i + 1
```

we're going to run some tests and instead of runni
been using so far, I'm going to run them directly in
and the reason for that is we'll get more accurate t
timing through the Web browser interface that we
we let
your programs execute and the timings won't be v
in the Python shell. If you have Python installed, yo
directions on the website how you can do that, but
. I
just wanted to demonstrate how time execution w
ne
the execution of the expression 1 plus 1. So that's
numbers. Well see how long that takes and when w
et
back the result which is one plus one is two. That's
took. A little hard to read so it looks like 8.3 and the
the scientific notation. The negative 05 is where the decimal point is so what that is really looks like this. There are fou
dot and that's a value in the number of seconds. For the number of seconds it took to run is .00008 seconds. If we try the same thing again we'll try timing the exact same thing. We don't get exactly the same result like the timing's inaccurate. It depends on other things going on o
answer. But it's still a very small number. And if w
we try, a more complicated addition, it's still goin
number of microseconds. The actual processor ti
than that but there are some other things happen
clock and other things to do the timing. We'll see
u



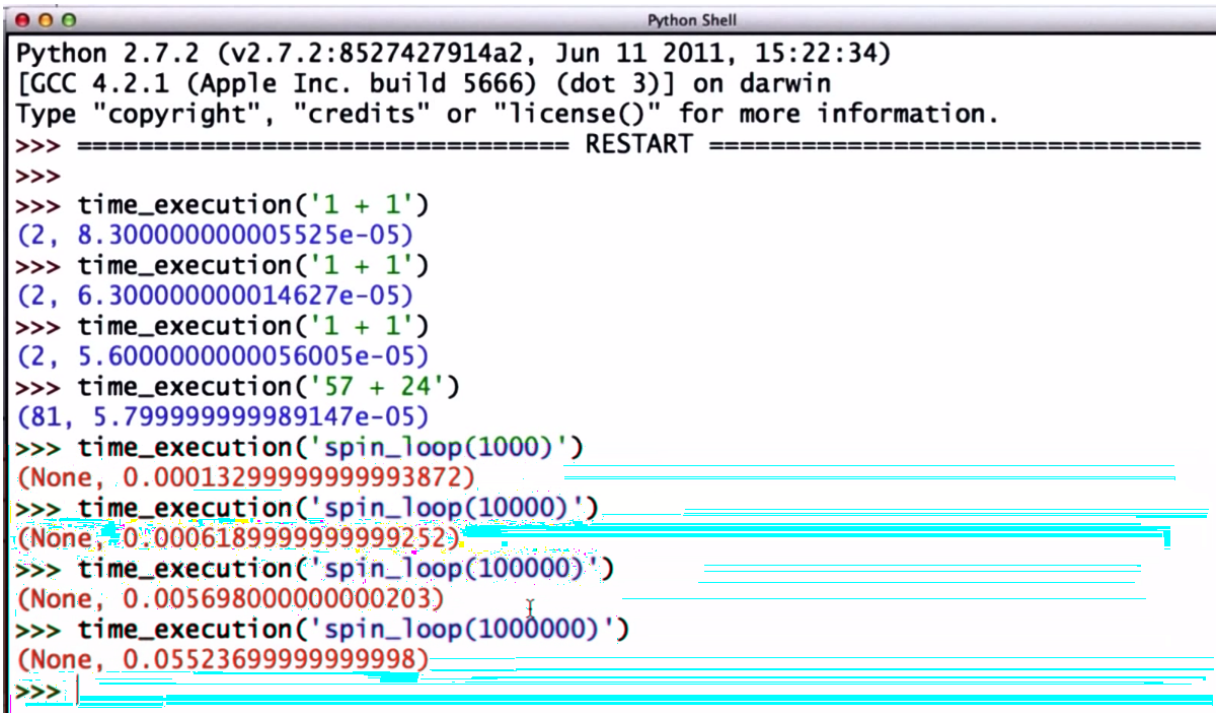
Spin Loop



So, to get a better sense of how timing works, I've defined a procedure spin loop. Spin loop starts by initializing the variable i to zero and then it goes through a loop n times, each time through the loop is just adds one to i. So that'll run for longer, we can, by picking the value of n, make it go through the loop any number of times. So, let's try that. So I'll try running the loop 1,000 times and now we get no result, and the time it took is 0.0001 seconds. So, about a tenth of a millisecond. Let's try it with a larger number of executions, so now I am going to increase the number of iterations by ten, so the value of n's multiplied by ten. And if we see that now, the time is, is higher, the time is increased to about six tenths of a millisecond and we can keep increasing the time, so let's try looping 100,000 times. And we see if the time increases. And the time increased by about a factor of ten. Between looping one more time. Let's go for a million. If we go for a million times, now we're up to 0.05. So, five hundredths of a second, or through that loop a million times. It's still much less than a second. What's important is, we can see that well, the time changes depending on the input. As we increase the input to spin loop, the time increases accordingly.

Run

```
1  import time
2
3  def time_execution(code):
4      start = time.clock()
5      result = eval(code)
6      run_time = time.clock() - start
7      return result, run_time
8
9  def spin_loop(n):
10     i = 0
11     while i < n:
12         i = i + 1
```



Programming Quiz: Predicting Run Time

This quiz will see if you understand execution time well enough to make some predictions. So, here's the code we had before that times the execution of evaluating a Python expression that we pass in as code. And we've defined spin loop as a y loop that goes through a loop that just adds one the number of times of the variable passed in. So let's try that in the Python interpreter. We're going to time an execution where what we're evaluating is calling spin loop, passing in some numbers. So we'll try 1,000 first. And what time execution does is return two values. The result and the run time. We only want the second one, so we'll index to get the second value out of the return result, and see that result. So that's what we get, so it took 0.0001 seconds to do spin loop a thousand. If I increase this to 10,000, I see that it takes 0.0006. Lets try 100,000, adding one more zero and I see now it takes 0.005. I'm going to write this a little differently, so we can see it more easily. So now, instead of writing out a 100,000, I'll do ten star, star, five, which is the same value as writing out a one followed by five zeros. And for good measure, we'll do ten star, star, six, which is a million times through the loop, and we see that, that takes 0.05 seconds. So here's the examples of timing loops. For your quiz, your goal is to estimate the expected execution time for a value running spin loop where the input of the loop is ten star star nine, that's one billion. And you should give your answer in the number of seconds. Of course it's not possible to get the exact answer, but you should be able to get a guess within about 20% of our answer to be correct.

Answer:

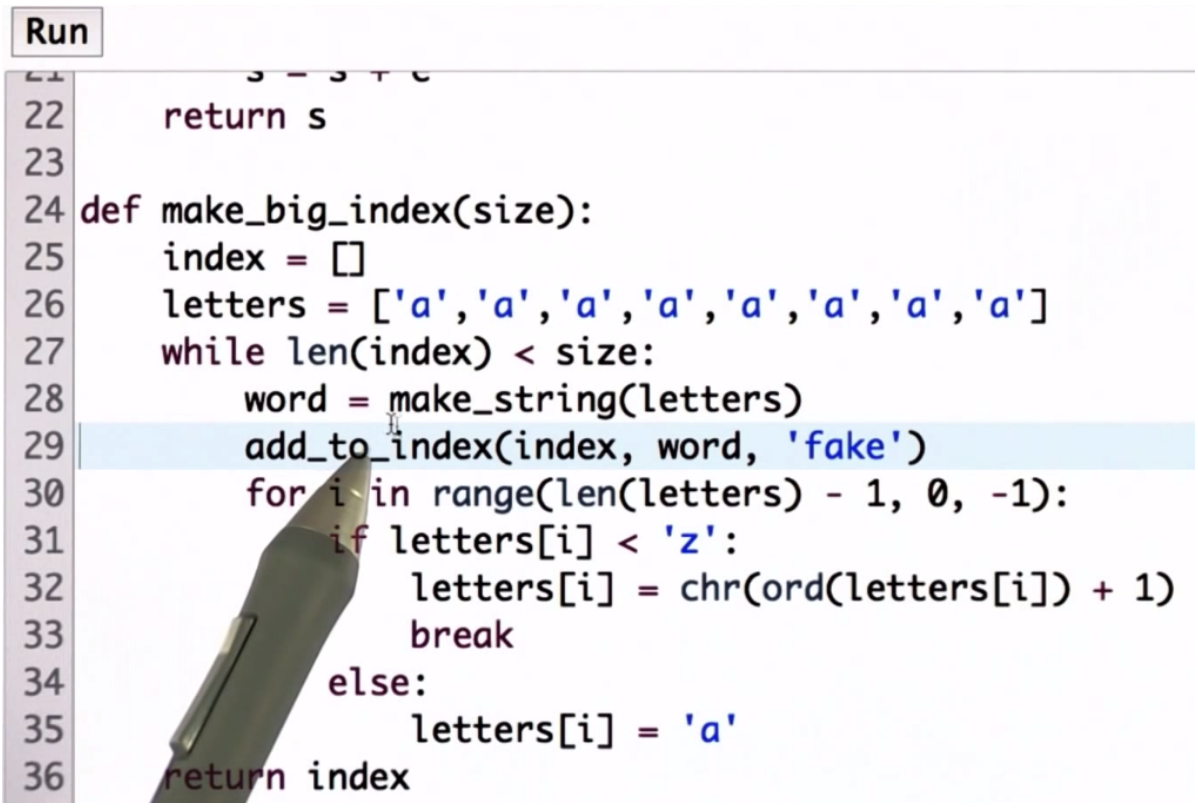
So here's the answer. We'll try it and see what we get. And while it's running, let's think about what we should get. So we can look at the examples that we did so far and try to make some predictions. So we saw when the value passed in was 10 to the 5, that the time it took to execute was 0.005. When the time passed in was 10 to the 6, so 1 million. We saw that the time to execute was 0.05, and now we're trying to predict 10 to the nine. If we look at the pattern here, every time we increase n by a factor of 10, the time also multiplies by a factor of 10. And, that's not surprising because the loop is going around ten more times. The times, number of times we go through the loop, scales as a factor of the input value n, so if we increase n by a factor of 10, the time will also increase by a factor of 10. And we see we have got our result now, so if we increase by another factor of 10, we would expect that this would also increase, that it would take about half a second to do 10 million.

If we increased by another factor of 10 we would expect the running time would also multiply by 10, so we'd be up to about 5 seconds. And if we increased by another factor of 10 which is the billion that we tried, we'd expect it to also increase by another factor of 10. Getting to be something around 50 seconds. So it's not exactly a 1,000 times what we had when we did spin loop passing in a million, but it's pretty close to that. Now it's taking almost a minute, 1,000 times this would be 54 seconds, so it's a little bit off from that. But very close, and if we tried it a few more times, we might get a slightly different result. Let's try is one more time. So we tried it again and this time we got 55.89 seconds,

pretty close to what we got the previous time. The important point here is that the running time depends on the value of the input to spin loop and it depends on in a linear way. As we increase the magnitude of n, the higher number of times through the loop, the running time scales linearly with that value.

Make Big Index

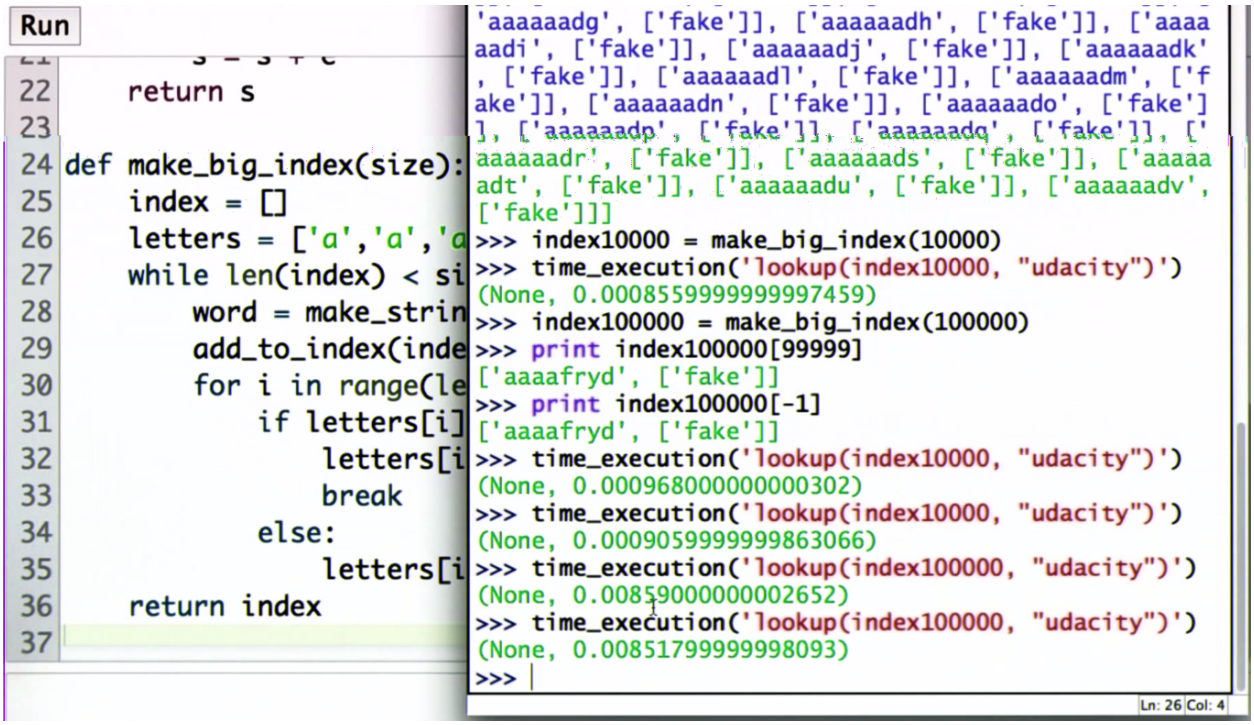
So we seen a few examples measuring the time of execution of toy programs, of a loop that does nothing. What we care about though is the time of execution of our index encode, so what I'm going to do next is write a program to be able to test the time of execution of indexing code. To get a good test we need to make a big index, we need to fill up an index with lots and lots of words, and we can do that by hand; that would take a lot of time and effort. So what I'm going to do instead, is to find a procedure that makes a big index. So what it does is take in, a size, and then it's going to, fill up an index with that number of words. To fill up the index with that number of keywords, we need to generate different words. So what I've done is created a variable called letters, that is initially is all A's. And as we go through the loop, we go through the loop size number of times. We keep making a new string. We add that string to the next, I'll explain what make_string does later. We add that string to the index, then we change the letters. And we're going to keep increasing the letter once we get to z, we wrap around. For now it's not too important to understand everything in this code, but I do want to walk through the code a little bit. So what this loop is doing is going through all the positions and the letters are A and we filled this up with eight different elements and we're going to go through those elements starting from the last one going backwards, so this range loop goes from the length minus 1 to 0, stepping by negative 1. We are going to check each letter. If the letter is less than z, that means we can increase and we are going to increase it using this code here and I'll talk soon about the code that turns letters into numbers, but what this expression does is get the next letter. So if the letter was an a the result after this will be a b, and we're going to replace the letter at that position in the letters list, with the next letter. If the letter is a z, we don't want to go beyond the alphabet, so instead, we're going to set that letter to a, and we're going to go back through the loop to find the next letter. Once we found one less than z, we break, we only need to change one letter. What the make_string procedure does, that we call here. Is just turn that array into one string. So it's going through the elements of P which is this list of letters that keeps changing. And concatenating all those letters into one string.



So the whole point of this is to allow us to easily make big indexes so we can run tests on different size indexes. So let's try this in the Python shell. First I'll show you what the result is when we use make big index. We'll start with a fairly small one, so I'm passing three is the size. What make big index gives us is an index with three keywords: aaaaaaaa, aaaaaaab, aaaaaaac and for each of them there's one url which is the name fake. If we passing in a bigger value, this will have an index with a hundred keywords, so we're going to passing a hundred is the size, we get this big index, and you can see it's starting to change the second from the last letter. To make sure that each word is a different word than the next. So, what we want to do now is look at how the time of executions take for different size indexes. So, let's make a really big index. So what we're going to do, we'll make an index of size 10,000. And remember, our concern is the time for lookups. It's the operation that's going to happen most frequently. So we're not timing the time to make the big index. Let's see what the time is to do a lookup, and so I'll time the execution of looking up in index 10,000 the keyword, and the word makes a difference. So, first let's try looking up the word judacity. Which, sadly, is not in our index. We'd need a much bigger index to get up to udacity. And the time of execution is shown here. So, it's 0.0008 seconds. So, still getting close to a millisecond, but still quite fast. Let's make a bigger index. This time we have a 100,000 keywords. going to take a little longer to make it, but we're not worrying about the time to make it now. What we care is the time to do a lookup.

So that took a long time. Let's see how, many entries there are, so we can look at the very last element in our index. And we can see got to aaaafryd, which I don't know to pronounce. Another way to see that, which we didn't talk about yet, we can actually index from the back using negative numbers, so

if you use negative 1 as the index, that will give us the last entry in the list. So, now we'll try doing a timed execution. We're going to look up in the 10,000 size index. And we'll see the time's pretty similar to what it was before, that time might vary a little bit, let's try it once more, and again, it's just under a millisecond. So now, we'll try, instead of the 10,000 index, looking up in the 100,000 length index, the same look-up, and we see that the time is now 10 times that, so it's now about 8.6 milliseconds whereas before it was 0.9 milliseconds. And let's, for consistency, try that again. We'll note that these timings vary a little bit, each time we do it. And there's lots of reasons why the timing varies. We're running lots of other things on the computer at the same time.



So it's not the case that we have total control over the processor and are running exactly the same thing every time. Because all of the other programs might be doing other things. The other reason the time can vary is where things are in memory. Sometime it's very quick to look up a value in memory, sometimes it takes longer. And we're not going to talk about the details of that. What matters is that, that the time's roughly the same, each time we execute it. And it really depends on the size of the input, in this case it's the size of the input table. So when we increase the size of the table to have 100,000 entries, it's about 10 times as slow as when we had 10,000 entries. So now let's have a few quizzes to see if you can guess how these timings work.

Quiz: Index Size Vs. Time

So the question is what is the largest sized index where we can do a lookup in about one second? So if you look at the executions we have, we've seen the time it takes with an index with 10,000 is .0009 seconds. The time with an index of 100,000 is .0085 seconds. And your goal is to predict what the

largest index that can support lookups that finish within about one second is. And your choices are

[ ] 200,000 keywords
[ ] 1,000,000 keywords
[x] 10,000,000 keywords
[ ] 100,000,000 keywords
[ ] 1,000,000,000 keywords

Answer:

So this required some guesswork, since we hadn't actually run any examples this large. But if you understood the examples we saw with the smaller inputs, you should have been able to guess that it would take about a second to do a lookup for a table with ten million keywords. And the reason for this, if we look at the previous executions, we saw, based on the keywords in the index, the time increased linearly. So, when we had 10,000 keywords we measured the time as 0.0009 seconds. When we had 100,000, we measured the time as 0.009 seconds. We didn't measure any times beyond that. The suggestion from looking at this is, well when, when we increased the number of keywords by ten, the time also increased by a factor of ten. So maybe if we could guess, that if we had a million keywords, it would probably take about 0.09 seconds, to do a lookup. If we had ten million, the time would take 0.9 seconds, which is close to one second. If we had a

Quiz: Lookup Time

So, for this question, your goal is to guess the expected time to execute the query. Lookup index10M where the string is eight a's. Where index10m, 10m, is the index created by make_big_index passing in 10 million at this size. And I should warn you this is a bit of a trick question. So, think careful about what make_big_index does and what lookup does. And here are the possible answers. So, it could be 0.0 seconds. It could be 0.1 seconds. It could be 1.0 seconds, or it could be 10 seconds.

[x] 0.0 s
[ ] 0.1 s
[ ] 1.0 s
[ ] 10 s

Answer:

So the answer is actually zero seconds that this will take very little time. It will be exactly no time but very close to zero seconds. And the very reason for this is because of both the way make_big_index works. Which creates an index where the words start with four A's, and we keep increasing the letters as we go. So that means the index created by make_big_index passing in any number. The first entry in that index will always be aaaaaaaa, and so when we do the lookup, the way lookup works, it goes

through the entries in order, and it's going to find the first entry right away. So the point I want to make with this quiz is that the execution time depends on both the size of the input, and sometimes it depends on the actual input. So here the size of the index is very big, because we looked up a word that's at the very beginning of the index, the lookup is very fast.

Quiz: Worst Case

What we usually care about when we analyze programs is what's called the worst-case execution time. And that's the case where the input for a given size takes the longest possible time to run. So for our lookup, the worst case will be either the very last entry in the table, or a keyword that doesn't exist in the table at all. Let's look at the code for lookup to understand more why the time scales the way it does, and think about what the worst case execution time and what the average case will be. So, to get a better understanding of the results we've seen, let's look at the codes that we wrote for the index. This is the code that we finished in the last unit. And we have the code for lookup. So, what lookup is doing, it's going through a loop. Each of the entries in the index. And remember that index is a list, so it's going through that list for each element, it's checking, does it match the keyword. So this is our index structure, right? It's a list of elements.

If we had 10 million of them, it's a very long list with lots of elements. But each element is a list itself, where we have keyword followed by the list of URLS where it appears. And what lookup is doing is going through those entries. The number of times we go through this loop depends on the size of the index. The size of the index is the maximum number of times through that loop. And it also depends on, well, if we find the keyword early, then we're done. The other code that you see here, that's relevant to this, is the code for add_to_index. And the reason that's relevant is, we want to know what the structure with index is after we make our big index. And what add_to_index does is also loop through all the entries to find if one already exists, then it adds that URL. If one doesn't exists, it adds the new entry at the end. So this means that the first entry we add will always be at the beginning. That's why the aaaa, with eight a's is first. And the last one that we add will be at the end. So now, we'll have a quiz to see if you understand what it means to be the worst case input and how the code for lookup works. The question is, which of these inputs will have the worst case running time? So the choices are, doing lookup, passing in the index and the first word that was added to the index. The second choice is doing lookup, passing in the index, and a word that is not in the index. And the third choice is calling lookup, passing in the index, and is the second input passing in the last word that was added to the index. So, for the quiz, you should check all the answers that have the worst case running time.

[ ] lookup (index, first word added)
[x] lookup (index, word that is not in index)
[x] lookup (index, last word added)

Answer:

So their answer is both of the last two. Both of these will need to go through the loop here, the number of elements in index times. In the case where it's a word that's not in a index, the test will always be false, and will get to the end of the loop and return none. In the case where the last word is added, we'll still go through the loop, the number of elements times, but the very last time we'll find a match, and return that element. One of the assumptions in all of this analysis is that the time it takes to go through the loop doesn't depend on the actual keyword that's passed in. And that's assuming that this equal comparison takes the same amount of time depending on what the entry and the keyword is. That it doesn't matter what the keyword is the time to do this comparison is the same. And it turns out that for strings and Python that's the case. That we can do these string comparisons very quickly, because strings are immutable. That means that we don't need to look at all the characters in the string to compare two strings. That double equal for strings can be done in such a way that it doesn't need to look at the whole string. It knows that the strings were created as different strings. That means they're different strings. Or if they were created as the same string, they're the same string. So that's the reason why we say that all of the other operations in the loop have constant time. That they don't depend on the size of inputs at all. These are all very fast operations. What matters in terms of understanding the time is the number of times we go through that index. If you take the follow on course, which I hope you will, we'll understand how to analyze algorithms in a more formal way. For now our goal is to develop an intuition, and for that intuition the important thing to understand is the running time depends on the number of times we go through this loop. Everything that we do inside the loop is constant time. It's not affected by the size of the elements.

Quiz: Fast Enough

So now I hope you understand the running time of look up well enough to be able to answer this question. This is definitely a fairly subjective question. And the question is, is our look up fast enough? So the possible answers are yes, that it is fast enough, that it depends on how many keywords there are in the index, that it depends on how many url's there are, meaning the number of different pages that we crawled in our web crawl. That it depends on how many lookups we expect to need to do. Or no; that it's not fast enough.

[ ] Yes
[x] It depends on how many keywords there are
[ ] IT depends on how many URLs there are
[x] It depends on how many lookups there are
[ ] No

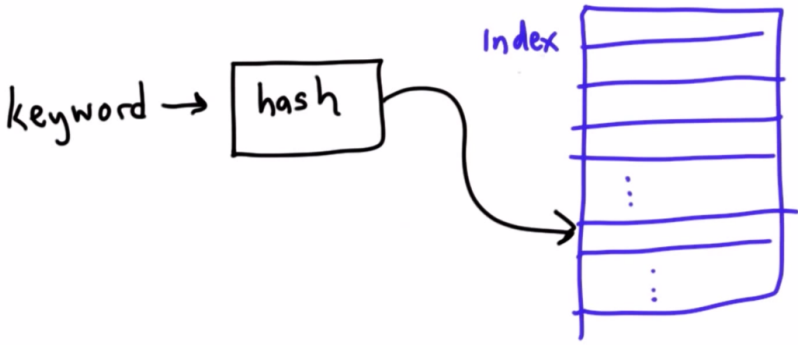
Answer:

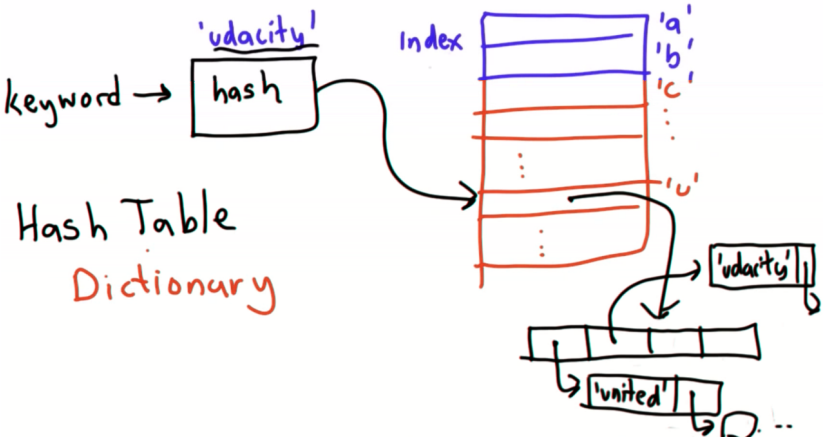
So the best answers for this question are that it does depend on the number of keywords. If we have very few keywords, well then, our lookup's fast enough. We're probably going to to have more keywords than that, so we want our lookup to be faster. The third possibility is not actually true. So, it doesn't depend on how many URLs there are. And the reason for that is we're doing a lookup based on the keywords. The number of times we go through the loop and look up depends only on the number of elements in index Which is the number of keywords, and then the size of the URL list for each keyword doesn't impact the time it takes to do a lookup. So it doesn't depend on the number of URLs. Certainly, if we have more pages, we're likely to have more keywords, so that would increase the size of the index, but just the number of pages itself doesn't affect the time for lookup. It does depend on how many lookups we want to do. So if we only want to do one lookup and that lookup takes a few seconds, well that's okay. It's probably not worth spending a lot of time to make that one lookup faster, but chances are we're going to want to do lots of lookups. Certainly if we wanted to have a real search engine, we'd want to be able to surf millions of lookups every day or in the case we will, many billions of lookups everyday, every hour. So, if we want to do lots of look up, we want look up to be faster. So, I think we have good reasons to want to make our look up faster and we're also going to learn some interesting computer science concepts, by trying to do that.

Making Lookup Faster

So what do we need to do to make lookup faster? Well let's think about why it was so slow, right? The reason it was so slow is that we were doing this for loop, we were going through all the elements in order, and we're checking if they match the keyword, right? And we had to do this for the entire index, for an entry, for a keyword that's not in the index, To determine that it's not there, we had to go through the whole index. This is not how we use indexes in real life, right? If you're looking for a word in the index of a book, you don't have to look through every single entry to see if that word exists. You can jump around, and the reason that you can jump around is because the entries in the index, are sorted, they're sorted in alphabetical order, so you know where that entry would belong. You just need to find the right place and see if it's there. So we can do that with our index instead of having our index kept in arbitrary order. If we kept our index in assorted order, then we could find the place where that entry belongs and look for it. Sorting is a very interesting problem. It's something we're not actually going to talk about more in this class. We're going to do a different way of doing that. What we're going to do is find a way to find where the entry should be that doesn't require actually keeping all the entries sorted. What we want is something that will allow us, given a keyword, we're going to have some function that tells us where it

belongs. We're going to call that a hash function. That tells us where in the entry to look. And so instead of having to look through the whole index, the hash function will tell us where that entry belongs. So what we need for this is some function that's going to take a keyword, map it to a number And that number is the position in the index where that number belongs. We could do this lots of different ways. One simple thing would be to think, well we know alphabet. This is more like the way an index for a book would work, and we're going to have for each entry and index, we'll have based on the first letter, we'll put all the entries that start with that first letter in the same place. So, if we're looking for a key word that starts with u, that prefer hash would tell us to look in the place where all the words that start with u are. And then we'd only have to look through the words that start with u. So this would allow us to do a lookup much more quickly than looking through the whole index. This isn't quite the best way to do things. If we made our places based on the letter, well, then we have a problem if we have two words with the same first letter. You generally expect to have more than one word that starts with the same letter. So instead of having just an element here for each position, we're going to have a list of elements that would be all the words that start with u So when we look up the word udacity, we look in the entry for u, and if the word that's there doesn't match then we know udacity isn't in the index. There are lots of problems with this.





The first problem is well, there might be more than one word that starts with u. So have is a list of entries We often call this a bucket. S with u that would be in this position. So, instead of index, now we're going to have a list of entries, and which is a list of entries that are in the right positio entries that start with u, and that would have all the different entries that start with the letter u in that bucket. So this i to look through all of the words in index, we just need to find the position that starts with the right letter. That's got a bucket of all the words that start with that letter, and then we just need to look through that bucket. This works okay, but this doesn't really scale very well. At best, if we have you know, ten million words well now instead of having ten million entries to go through, we need to go through ten million divided by say 26, if we have 26 letters. It's not making things much faster. It's making things maybe at best, 26 times letter. That assumes that all of the buckets are the same size. Certainly if we make the buckets based on the first letter, that's not going to be the same size. If the

words are a typical English words. We're going to have many more words that start with s or t, say than start with u. So, we want to fix those two problems. We want to be able to have more buckets. So we're not going to just use the first letter, we're going to use some function on the whole word that tells us where it belongs. And we're going to try to make that function distribute the words fairly well. So the structure that I've described is what's called a hash table. This is a very useful data structure. It's so useful that it's built into Python. There's the Python type called a dictionary. Which provides this functionality. At the end of today's unit, I'll explain how the Python dictionary works, and how to use it, and we'll modify the search engine code to use dictionary instead of the lookup table that we built, but before we do that, we are going to implement it ourselves. We are going to make sure that we understand how the hash table works by writing all the code to do it ourselves and then we'll switch to using the built-in Python.

Quiz: Hash Table

So now we're ready for a quiz to see if you understand the goal of the hash table. So the question is if we have b buckets in our hash table, and we have k keywords, and we should assume that k is much greater than b, that there are more keywords than we have buckets. The question is which the properties should the hash function have? And remember what the hash function is, is it's a function that takes in a keyword, produces a number, and what that number does is gives us the position in the hash table which is the bucket where that keyword would appear. The first choice is output a unique number between 0 and k minus 1, so each keyword maps to its own output number. The second choice is output a number between 0 and b minus 1. The number of buckets that we have. The third choice is that it should map approximately k divided by b, of the keywords to bucket 0. That means for that number of keywords, the output of the hash should be 0. And it should map to the first bucket. So the fourth choice is map approximately k divided b of the keywords. To bucket b minus 1. That's the last bucket. And the final choice is it should map more of the key words to bucket 0, then it maps to bucket 1. So check all of the properties that we would like the hash function to have.

[ ] Output a unique number between 0 and k-1
[x] Output a number between 0 and b-1
[x] Map approximately k/b keywords to bucket 0
[x] Map approximately k/b keywords to bucker b-1
[ ] Map more keywords to bucket 0 than to bucket 1

Answer:

So, the answer is many of the properties are desirable. The second property is desirable. So, the first property is not desirable. That if the hash function outputs a unique number for every keyword, well then, the range for the hash function would be very large. We'd need a huge amount of memory to store the hash table. And the number of buckets would be the same as the number of keyboards.

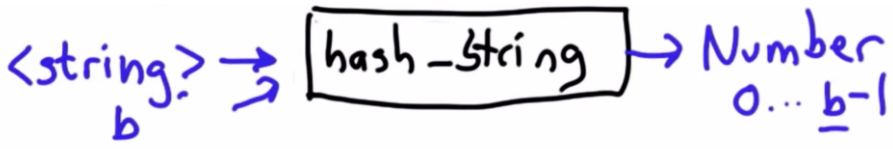
That's not going to work very well. What we want is the number of buckets to be b, so that means that the output of the hash function should be in the range between 0 and b minus 1. That will find the element of the list that corresponds to that bucket. So, we do want the second property. The third and the fourth properties say that we want the number of keywords in each bucket to be approximately the same.

So, if there are k keywords and we're fitting them into b buckets, if we want the buckets to be approximately the same size, then each bucket should have approximately k divided by b keywords. So, both the third and the fourth property are true. The final property says that we should have more keywords in bucket zero than in bucket one. There's no reason that we would prefer that, so you, you might think it would be better to have more keywords at the beginning than later on, that's not the case with a hash table. Remember, what our hash function does is, it tells us right away which bucket to go through. We don't have to look through the earlier buckets. This is different from the list index that we started with. Where the first one is the fastest one to find. And the last one requires going through all the elements. For the hash table, that's not the case. There's no reason to prefer having more entries in bucket zero to bucket one. What we really want is to try to have all the buckets have approximately the same number of keywords.

Hash Function

So let's try to define a hash function that has these properties. And what we want the hash function to do is to take a string as its input, we'll call the hash function hash_string, and it'll produce as output a number between 0 and b. So, we also need another input to our hash string, which is going to be the size of the hash table. So
that'll be the second input is the size of the hash table, the number of buckets. What we haven't seen yet, that we're going to need for this function, is a way to turn a string into a number. Python provides an operation to do that. It's called ord as an ordinal, and what ord takes as it's input is a one letter string and produces as it's output a number. And actual mapping between strings and numbers is not so important. We just want something that is going to map different string to different numbers. There' takes in a number, and outputs the one-letter string that corresponds to that number. And the property these functions have is their inverses. That if we take the character corresponding to the ordinal corresponding to any one-letter string. We'll call that alpha. What we get as a result is the same alpha that we passed in. So let's try a few examples in the python interpreter to see how ord and chr work. So we'll print ord of a, and when we run that we see we get the number 97. If we try print ord of capital A. That's different. We get 65. And if we print ord of B, we get 66. So, the numbers are sort of sensible. B is higher than A. The lower case letters have different ordinals than the upper case. So, if we try a lower case b, we should expect to get 98. And that is indeed what we get. And these are the numbers based on the ASCII enchar, character encoding, what the actual numbers are, are not very important for us, other than that we get different number for different letters. So we'll be able to use the results of ord to make different strings hash to different values. And just to show that there are

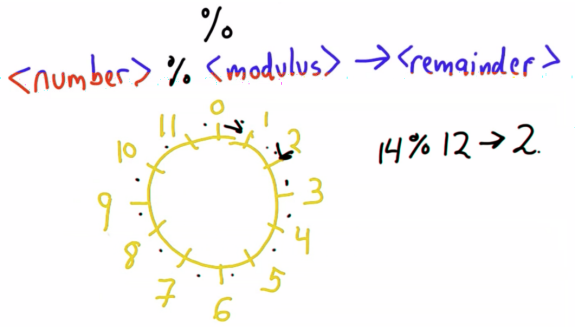
inter, inverses. If we do ord of u, and then chart of that, what we get back is the single letter string u that we started with. The limit of ord is it only works on one-letter strings. If it provided a mapping from any string to a number that would be useful for a hash table. Well, then we'd be done. But it doesn't do that. If we try running it on a multi-letter got a string of length 7. So have a way of converting st The other property we nee

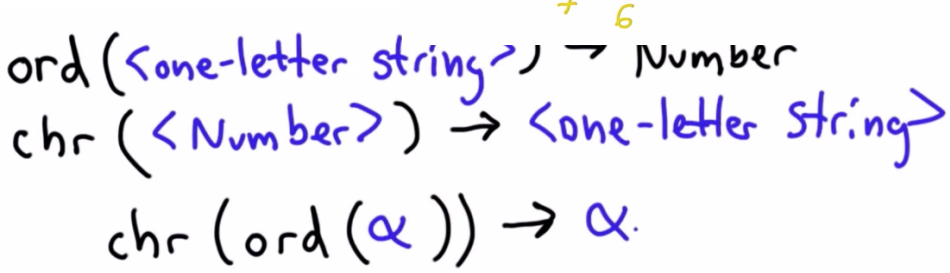


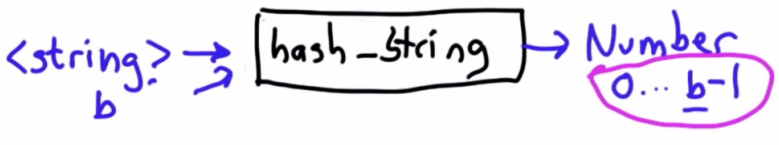
between 0 and B minus 1. We need it to be in that range, because we're going to use that to index the list, to find the bu

Modulus Operator

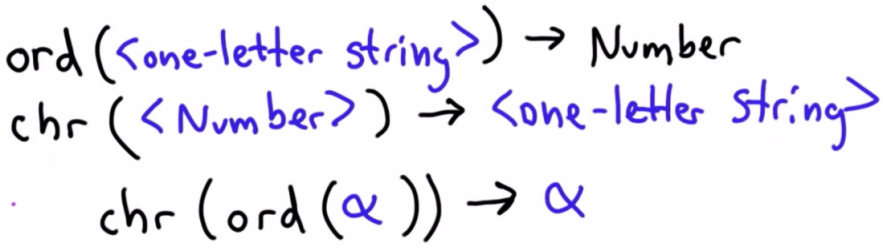
So to get our outputs in the correct range, we're going to use the Modulus Operator. And this is written with a percent sign. It's usually the Shift 5 on most keyboards. What modulus does is takes a number and maps it to the range based on the remainder when you divide that number. So the way modulo arithmetic works, is like a clock. So if we think of having a clock with were to evaluate 14 modulo 12, what that would mean is, well, if we start from 0, and we make 14 steps, we're going to 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, as the remainder we would get by dividing 14 by 12. And that's the result of 14 modulo 12, is 2. So now we're going to have a few quizzes, to make sure you that understand the Modulus Operator, as well as the org and char operators we introduced.







Programming Quiz: Modulus Quiz



So for this quiz, your goal is to predict the value of each expression. Try to figure out the answer yourself. You can definitely try evaluating in the Python interpreter to check that it's correct. So, the first expression is 12 modulo modulo ord the character a. And you should see if you can solve this, without actually figuring out what the value of ord a is. And the third question is ord of z plus 3 modulo ord of z.

Answer:

Here are the answers. So, for the first question, we had 12 modulo three and 12 is equal to three times four. So, that means, if we divide that means, the result of 12 modulo three is zero. For the second question, we could solve this by figuring out what ord of a is. But we don't actually need to. We know that the value of ord for 'a' is always going to be the same every time we do it. So this is saying, the same value modular the same value, the result is always going to be zero. Because anything is divisible by itself with no remainder. So that answer's also zero. For part C, well now we've added three. In order to know what the result is here, we've got to know whether ord of 'z' is greater than or less than three. As long as ord of z is greater than three, well that would mean ord of z plus three modulo ord of z must be three, because the remainder will be what we added to ord of z, since ord of z by itself is always divisible by ord of z. And we do know that ord of z is greater than three. So that means the result will be 3, so let's try that last one in the Python interpreter. We're going to print the value of ord of z, plus three, modulo ord of z, and when we run that, we see the result is three, and I want to point out that the parentheses here are actually important, if we didn't have them We tried the order Z, plus three order Z, then we get the result 125. And the reason for that is, the grouping here is going to group three mod or Z and add that towards, so it's a different value, and the reason we get 125 is the value of ord Z by itself. Is 122. So it's 122, which is ord of z, plus three mod ord of z, which is three mod 122, which is three, which is why we got 125 when we evaluated this without the parentheses.

Quiz: Equivalent Expressions

So we're going to find lots of good ways to use both modulus and or Intera operators. So lets do one more quiz that's going to be a little trickier. What I want you to answer is which of the expressions below are always equivalent to X. So I have exactly the same value of X where X can be any integer between zero and ten.

[ ] x % 7
[x] x % 23
[x] ord (chr(x))
[ ] chr(ord(x))

Equivalent Expressions Solution

So the answer is, only the second and the third are equivalent. This is a little surprising. This was kind of a tricky question. So the reason the first one is not, if x is seven or greater, well, then seven mod seven has the value zero. That's not the same as the x that we started with. And that's the case also if we have eight. Eight module seven has the value one. Which is different from what we started with. When the modulo is greater than the possible value of x, and we said, x could be only between zero and x. The third question, when we map x to its character value, and then we take the order of that, we'll char an order inverses, so that's equivalent. You would think that would in the other direction and the reason it doesn't is because the input to ord must be a one letter string. If the input's not a single character, then ord produces an error.

So let's see that in the Python interpreter, if we print the result of ord where the input is a number and we said x was a number between zero and three. Well, that gives us an error. And it gives an error, because ord expects a string of length one, but the input was an integer. There is a function that allows us to turn numbers into strings. And that's the str function that takes a number and gives us a string corresponding to that number. So let's see what str of three gives us, that will give us the string three, we can't see in output here that it is actually a string, but it is a string, and we can actually use order on the result there. When we run this, what we get is 51, not the three that we passed in and that's because the ord of the character three is 51.

Quiz: Bad Hash

So now that we know about ord and we know about modulo, we're ready to define a hash function. So let's remember what our hash function should do. So the hash function takes two inputs. It takes the keyword, and it takes the number of buckets. And it's outputting a number between 0 and buckets minus 1, that gives us the position where that string belongs. And we've seen the function ord that takes a one letter string and maps that to a number. And we've seen the modulus operator that takes a number and a modulus and produces, as a result, the remainder that we'd get when we divide that number by the modulus. So now, we can use those to define a hash function. I'm going to start by giving you a really bad hash function. So let's define a procedure, and we'll call it bad_hash_string to make sure we remember that it's really bad. And it takes as inputs the keyword and the number of buckets. And we're going to output the bucket based on the first letter in the keyword. So we'll use ord on the first letter, and we'll use modulo the number of buckets. And that's our simple hash function. For the quiz, I want you to answer all the reasons why that's bad. And remember what bad_hash_string does is take the first letter in the keyword, turn it into a number using ord, and then find that value modulo the number of buckets. Here are the possible reasons, check all the reasons that apply. The first is it takes too long to compute. The second is there's at least one input cured for

which bad_hash_string will produce an error. The third is, if the keywords are distributed like words in English, then some of the buckets will get too many words; others will get too few. And the fourth choice, if the number of buckets is large then there's some buckets that will never get any keywords.

[ ] It takes too long to compute
[x] It produces an error for one input keyword
[x] If the keywords are distributed like words in English, some buckets will get too many words
[x] If the number of buckets is large, some buckets will not get any keywords

Answer:

The answer is, all choices are true except for the first one. The reason the first one's not true, is this really doesn't take long to compute. We only need to look at one letter and do a simple modular computation, that's very efficient. But the other three reasons are true and we'll go through each of these showing what happens looking at how things evaluate in the Python Interpreter. So the first correct reason is that it produces an error for one input keyword. When we write code we should think about whether it works for all possible inputs. And the one that's usually the trickiest to think about is the boundary case. For a string, that's often the empty string. So, if we pass in a string with no characters in it, which is a perfectly valid string, we'll then, when we try to index element zero, that would be an error. So let's see what happens when we try that in the Python Shell. So we'll try to evaluate bad_hash_string, passing in the keyword the empty string, which is a perfectly valid string. And lets see there are 100 buckets. And we do get an error, we get the error that the string index is out of range because we tried to access the character at position zero, but there is no character at position zero in the empty string.

So to understand the other two reasons, I have defined a procedure called test hash function. It takes three inputs. The first input is a function, so we can pass functions around, just like any other value, so what we're going to pass in for this, is the bad_hash_string function that we've defined, but we can also use it to test other hash functions, which we'll see later. We're going to pass in a list of keys, those are the keywords for the hash table, and we're going to pass in the size, this is the number of buckets. What we do in test hash function, is we're going to keep results as a list of the number of times each bucket is used. So initially, they're all zeroes. And we initialize it with zero times the size. We're going to use keys used as a list of the keys that have already been used. We don't want to count a duplicate key more than once. So now we're going to loop through the keys. We're going to check if a key was used already, and if the key was not used, then we're going to figure out by calling the hash function where that key would hash to.

So, if we passed in that hash string, that would be the function here. And we're calling that passing in the keyword, and the number of buckets. We're storing the result in the variable hv. And then we're

increasing the value of the element at results position hv, by what? And this is a shorthand syntax, means the same thing as doing a new assignment where we're assigning two results hv, the value currently end results hv plus one, and then we're adding the word that we just used to the list of keys used, so we don't use it again. This is similar to what we did in the web crawler, to avoid crawling the same page more than once. And if end will return the results. So what we'll have as the result of test_hash_function, is a list where the values in that list are numbers, giving the number of times a key hashes to that bucket. So let's try this with an example using the bad_hash_string function. So to test our hash function, we need some content. We need content that represents the kind of words that we think we're going to be using the hash table on. I've picked this one, which, perhaps is represented, perhaps not. And what's there at that link, is Gutenberg's text of The Adventures of Sherlock Holmes. And you can see from the scroll bar, it's quite long. So this is all the text there is. And so on. So, we're going to get all the words on this page using getpage. We're going to split them into words like we were doing in the crawler, and we'll store that in the [INAUDIBLE] words. And the length of that is over 100,000 words. Now they're not all unique, so the number of entries in our hash table will be smaller than that. But let's see how the distribution is for those words.

So we'll use the test_hash_function that we defined, passing in bad_hash_string, the words that we got from Sherlock Holmes, and we'll pick, for now we'll use size 12, definitely too small, but that'll give us a good sense of how the distribution goes for a small number of buckets. So now we have the result. Let's look at what the counts are. And you can see, we've got 12 entries, which corresponds to the number of buckets. And they vary quite a bit. The smallest one has only 754 elements. The largest one has over 2000. So the gap between the smallest and largest is nearly a factor of three. If our hash function was good, we will want these to be about the same size. Here's what that looks like graphically. We have our twelve buckets. The ones that are red, are too full. The ones that are blue, are not full enough. We would like this to be a fairly flat graph, distributing all of the words evenly between the buckets.

Programming Quiz: Better Hash Functions

So we've seen the basic looking at just the first letter does not work very well, doesn't use enough buckets. And it doesn't distribute the keys well. So now we're going to think about how to make a better hash function. So we want the same property we had before in the sense that it's one function that takes two inputs and that two inputs are the keyword which is a string. And the number of buckets, which is a number. Those are the two inputs and the output is the hash value and it's in the range from zero to number of buckets minus one. And our goal is for these numbers to well distributed, so we can have any number of buckets we want. The keywords will be spread evenly among the buckets, and every time we hash the same keyword we'll get the same bucket. So, we'll know quickly where to find it.

So in order to do better than we were doing before, we're going to need to look at more than just one letter of the keyword. If we look at just the first letter, we're not going to do better than having a limited number of buckets in a bad distribution. So, what we want to do is something that's going to look at all the letters of the keyword, not just the first letter. And based on all the letters, we'll decide their appropriate bucket. We saw that with lists, if we had a list of items, we could use the for loop construct to go through the elements in the list, like this. And this will go through each element in p, assigning it to the variable e and executing is whatever is in the block for each element of p. We can do something similar with strings, so if we have a string s, we can use the same construct to go through the characters in s.

So each time we go through the loop, the value assigned to variable c will be a one letter string corresponding to each character in the string in order. So first it will have the value a, and then value b, and then value c, and then value d. So this gives us a way to go through all the elements in the string. And if you remember how we turned single letter strings into numbers and modular arithmetic, then I think you know enough to define a much better hash function. So for this quiz, your goal is to define a function hash_string that behaves like this. It takes a keyword, a number of buckets, and it gives it number, identifying the bucket where that keyword will belong. But instead of just using the first letter in the keyword, it's going to use all the letters. And produces as output, a single number that represents the bucket where that keyword belongs. And your goal is to do this in a way that depends on all the characters in the string, not just the first character. There are lots of ways to do this, so we're going to specify exactly what you should do.

So what we want to do is make the output of hash string a function of all the characters. And we can think of that with modular arithmetic, that if we have a circle which is the size of the number of buckets, so this is going to go from 0 to the buckets minus going to start at 0 and for each character in the string, we are going to go around ord of that character, distance around the circle, and we are going to keep going. So each character we are going to go some distance around the circle, the circle can be any size depending on the number of buckets and we're going to keep going around the circle using modular arithmetic for each letter. And as we keep going, we can go lots and lots of times if it's a big word. Wherever we end up is going to be the bucket that we use. And let's say that's position 27, who knows where it is. So that's the idea and I'll give you a couple of examples so you know if you're implementing it the right way.

Let's suppose we have as our string the single letter a, and we have 12 buckets. Well in that case we're going to have 12 buckets, so we wouldn't have 27 here. This is going to be, do is go around the circle eight times, because 97 is 12 times 8, and one more, because 97 is 12 times 8 plus 1. You don't actually have to go around the circle if you use the modulo that's what it will give you. And so the hash value for a should end up in bucket one. As another example, if we tried hash string where the string is the single letter b, also with 12 buckets, well that's going to also go around the circle eight times because the ord of b is 98. But it's going to end up in bucket two, since that's 96 plus 2, which will end up in bucket two. If we change the number of buckets, the results will be different, so let's say, instead of

having 12 buckets we have 13, and we tried hash_string on the single letter string a, with 13 buckets. The result there should end up being 6, and the reason it's 6 is because 97 is equal to 13 times 7 plus it to just work on single letter strings. With single letter strings, we are really getting the same thing we defined before, where it gets interesting is where we have multi letter strings. So lets look at a few examples like that. So where we can see the difference between what you are going to find at hash string and the bad hash string we had before is once we have longer strings. So suppose we tried the string au, the ord of a was 97, and that, when we went around the circle, ended up at location 1. The ord of u, I can tell you, is 117. When we add those two, we get 214. And modulo the number of buckets, which is 12 here, we should get 10. So that string with two symbols doesn't end up in position one where the a would end up. It ends up in bucket ten, and as another example, we won't work through the details on this one, but you'll be able to test if your result is correct. If you try the string udacity, it should end up in the bucket sorry I forgot the input size, this is very important. We need two inputs to hash string, both the string and the number of buckets matter. So here, we're also assuming this. And we want it to take these two inputs and produce a number that tells us what bucket that keyword should be in.

Answer:

So, this is a pretty tough question. If you weren't able to get it yourself, that's okay. What I'd encourage you to do is stop in the middle of the explanation, once the part where you got stuck on makes sense, and see if you can finish it yourself. So, what we need to do is define a function, so we're going to define a function hash-string, and it takes two inputs, the keyword and number of buckets. And we're going to keep track of where we are in the circle. We need to introduce a new variable to do that, and we should start at position zero. So, we'll initialize that variable. We'll use h to represent the hash, and we'll initialize that to zero.

Now, we want to go through the characters in the key word. So, we'll have a for loop that goes through each character in keyword. And for each character, we want to add to the hash. So, we're going to add to the hash, the value of that character. We could do the modulo here, so we could, at this point, use modulo buckets. We have to be careful to have the parentheses here. If we just have the modulo buckets here, we wouldn't get the right result. because it would do ord c modulo buckets, but what we really need to do is modulo buckets, the sum that we get from h and ord c. And then at the end, we're going to return the hash value. We could, instead of doing the modulo each time here, we could do the modulo just once at the end. That would be computing a big number if we have a really big string, and then, at the end, compute our modulo buckets. Either way should work. This way's a little better, in the sense that if our string is very long, we would have to compute a really, really big number, which gets to be more expensive. And we might even run out if it's a super long keyword. So, it's better to do the modulo here. We get the same result either way though. Let's try this in the Python interpreter.

So, here's the code that we wrote out on the sketchpad. We have our variable, h, which is going to keep track of the hash value. We're going to go through all the characters in the keyword, adding each one into the hash value, modulo the number of buckets. So let's try the examples. So a, with 12 buckets, hashes the bucket one as we expect, and if we look at b. And hash just the bucket two, also as we expect. So, one other thing we should try, if you remember, when we had the really bad hash string function. One of the many problems it had was it didn't work on all strings, in particular, did not work on empty strings. Do you think our hash string function here will work on the empty string? Try to guess what the result should be before I run it. And then I will run it. And you see the result is zero. So, no error. And it makes sense that the result is zero. We start with h is zero. When there are no characters in the string, we don't go through this loop at all. So, h is still zero when we return. And let's also try the longer example. When we hash the string udacity with 12 buckets, we get bucket 11. We should be able to increase the number of buckets. So, let's increase the number of buckets. Let's suppose we had 1,000 buckets, and we get bucket 755. This isn't enough to convince us that our hash string function is distributing all strings well between all buckets. But at least we're getting a, a fairly large number that says we might be using all the buckets.

Testing Hash Functions

So let's test our new hash function. See if it does better than the bad string hash function we defined earlier. We're going to use the same test hash function we defined before, that takes the function as input. So we can pass in either the original bad string hash function, or the new hash string function that we hope will work better. A list of keys, and the size, computing t up in each position. So let's try that again, we from Sherlock Holmes. And so we'll initialize w Adventures of Sherlock Holmes, that we load function, bad string hash, and obtain the cou like. So it's a pretty bad distribution, as we sa in one bucket. Now, let's try it with the new ha in plain hash string. We're using the same wor distribution now. It looks a lot better, right. Th values over the highest one would be 1,363 in what we had before. With a bad hash string function we can see the size of the buckets varies a great deal. And we have popular, some that are not popular enough. With the new hash function we have much less variance. Still not perfect. We'd like to have all the bars really be as close to the same as possible, but it's really close. So this is working pretty well. The other thing we can try is having more buckets. So lets try this one, we are doing the same thing but this time with a 100 buckets instead of the results when we have 100 buckets, are pretty good, but certainly not perfect. We have buckets as small as this one, that has a 104, and as larger, this one that has a 197. So almost twice the size of the smallest bucket. It's certainly a hard problem to build a better hash function. People put a lot of effort into building good

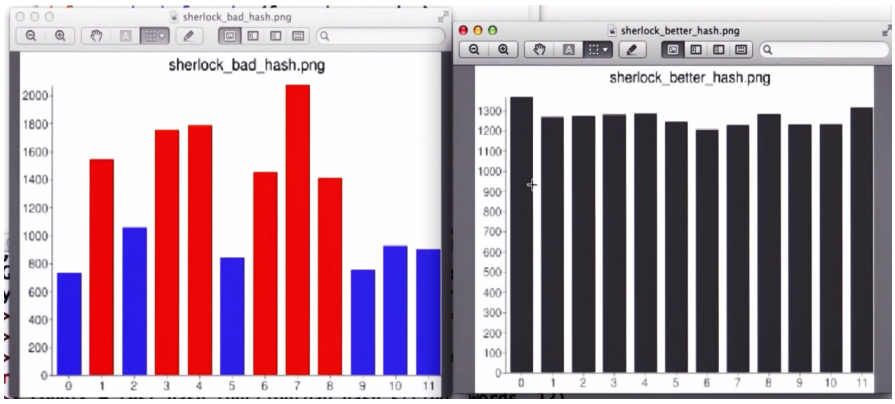
hash functions. As your tables get larger, it's very important to both have the hash function be efficient. Our hash string function is not that great, because it does take a long time to execute. If the string gets longer we have to go through that loop once for each character. And so there are better hash functions available.

We're not going to look at those in more deta
documents about more interesting hash func
before we go on to actually implementing in a
t

```
def test_hash_function(func, keys, size):
    results = [0] * size
    keys_used = []
    for w in keys:
        if w not in keys_used:
            hv = func(w, size)
            results[hv] += 1
            keys_used.append(w)
    return results
```

Ln: 35 Col: 0

```
Python Shell
Python 2.7.2 (v2.7.2:8527427914a2, Jun 11 2011, 15:22:34)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
>>>
```



```
def test_hash_function(func, keys, size):
    results = [0] * size
    keys_used = []
    for w in keys:
        if w not in keys_used:
            hv = func(w, size)
            results[hv] += 1
            keys_used.append(w)
    return results
```

```
Python Shell
[725, 1509, 1066, 1622, 1764, 834, 1457, 2065, 1398, 750, 1045, 935]
>>> counts = test_hash_function(hash_string, words, 12)
>>> print counts
[1363, 1235, 1252, 1257, 1285, 1256, 1219, 1252, 1290, 1241, 1217, 1303]
>>> counts = test_hash_function(hash_string, words, 100)
>>> print counts
[152, 135, 113, 142, 145, 153, 123, 114, 125, 126, 146, 136, 147, 120, 141, 134, 142, 144, 1
40, 135, 126, 104, 136, 136, 131, 153, 142, 169, 136, 145, 158, 149, 175, 141, 142, 175, 145
, 157, 153, 153, 168, 148, 182, 154, 177, 163, 165, 138, 163, 157, 149, 154, 166, 173, 159,
162, 185, 158, 165, 172, 171, 159, 139, 152, 167, 150, 143, 151, 154, 174, 129, 184, 164, 17
6, 145, 159, 161, 149, 151, 163, 163, 151, 170, 156, 197, 160, 172, 142, 189, 141, 159, 155,
128, 139, 126, 164, 161, 156, 140, 163]
```

Quiz: Keywords And Buckets

So, lets assume for now that we have a perfect hash function. That it distributes keys evenly across all the buckets. And then the question is which of the following will leave the expected look up time for a given keyword essentially unchanged. For the first choice is we can double the number of keywords, without changing the number of buckets. The second choice is we can keep the number of keywords the same. But double the number of buckets. The third choice is, we can double the number of keywords and double the number of buckets. The fourth choice is we can halve the number of keywords, keeping the number of buckets the same. And the final choice is we can have half as many keywords, and half as many buckets. And the question is, which of these five, and there could be more than one that's correct, Will essentially leave the time it takes to look up a keyword unchanged.

[ ] Double number of keywords, same # of buckets
[ ] Same number of keywords, double # of buckets
[x] Double number of keywords, double # of buckets
[ ] Halve number of keywords, same # of buckets
[x] Halve number of keywords, halve # of buckets

Answer:

So there are two correct answers, the third one, and the fifth one. And this is why a hash table is such a great advance over the linear index, is that we can double the number of keywords, and double the number of buckets, and the lookup time stays the same. With the linear index, if we double the number of keywords for each look up, we need to go through the loop once for each keyword. If the keyword was near the end or one that wasn't in the table, the time to look up the keyword would double as we double the number of keywords. With a hash table if we also double the number of buckets when we double the number of keywords, well then the number of keywords in each bucket stays the same. We're dividing the keywords evenly between buckets. So this is the number, the number of keywords per bucket is the number of keywords divided by the number of buckets. Of we double both, that number stays approximately the same. The time to look up only depends on the number of keywords per bucket. The time to find the bucket is very fast, right? We just need to run the hash function, find that element of the list.

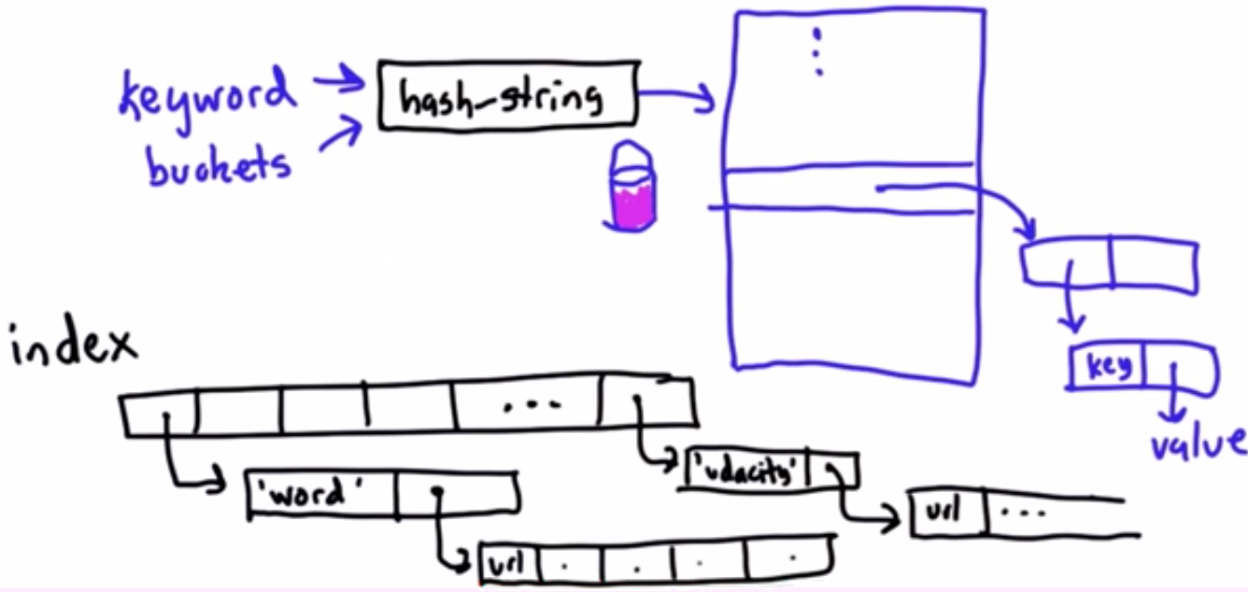
Both of those don't depend on the size of the list, how long it takes to do that. And then we have to look through the bucket, the size of the bucket, we have to look through each element in the bucket one at a time. So if we keep the number of keywords per bucket the same, the lookup time stays essentially the same. So that's the great property that hash tables have. If we double the number of buckets, as we double the number of keywords, the expected look up time doesn't change. For the other possibilities, well if we double the number of keywords and keep the same number of buckets, that's going to get slower because the number of keywords per bucket will approximately double. So

it's going to take about twice as long for each look up.

If we keep the same number of keywords, but double the number of buckets, well then it's going to actually get faster. We'll have the same number of keywords, double the number of buckets. So, this value will be approximately half of what it was before. The expected lookup time will be about half of what is before we doubled the number of buckets. If we have the number of keywords keeping the same number of buckets, well that has essentially the same effect. The average number of keywords per bucket will be half of what it was before, so the expected lookup time will be about half of what it was. And finally if we have both, well that's going to keep the ratio the same so the expected look up time will be about the same. So that's why these two are the correct answers, that those are expected to leave the lookup time essentially unchanged.

Programming Quiz: Implementing Hash Tables

So I hope everyone understands the main idea behind the hash table now. Our goal is to map a keyword and a number of buckets. Using our hash-string function to a particular bucket and that bucket will contain all of the keywords that map to that location. So now what we're going to do is try to actually write the code to do this. We're going to start from our index that we wrote for the previous unit, but try to figure out how to implement that with a hash table instead. So the first question is, how is this going to change our data structure? So this was what we had before, if you remember we had our index was a list of keywords, we had a list of entries, and each entry was a pair, which was a keyword. And the second element of the pair was the list of the URLs where that word appears, and we would have each word in the index as its own entry with its own list of URLs. So this was the data structure that we used last class. Now we want to change things to implement a hash table. So, I want you to think about what data structure we'll use and we'll make that a quiz to decide a good data structure to use to implement the hash table. So the question is which of these data structures would make most sense to implement the hash table index? The first choice is a list. Where the elements in the list are. A list where the first element is a word and the second element is a list of URLs where that word appears.



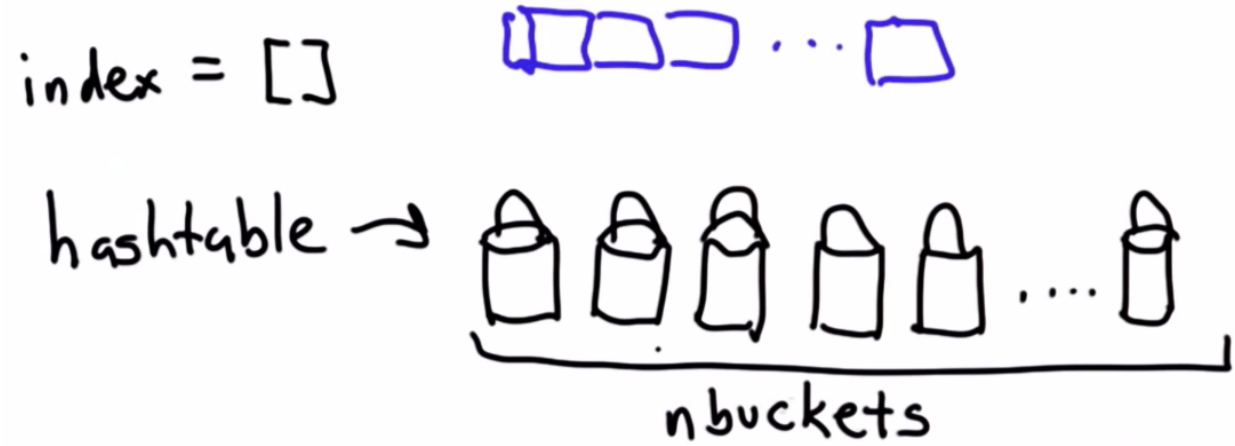
The second choice is a list where each element in a list is the list itself where the first element is a word and the second element is a list of lists where each element in that list is a list of URLs. So, the third choice is a list. Where each element is a list. Where the element lists themselves contain lists. Where the element lists of the element list are a list of the word, all by a list of all the URLs for that word. So, we have three nested lists for the first three. For choice four, we have a list where each element in the list is a list. Where within the element list, there is another list, which is a list of words, followed by a list of URLs. And for the final choice, we also have three nest of lists, where each element is a list, where the elements of that list are a list, that are word followed by a list. Where each element in that list is a list of urls. So which one of these would be the best structure to implement a hash table?

Answer:

So the answer is the third choice. These data structures are getting complicated enough that they're fairly hard to read. We need to think about what components we need to represent the hash table. And the main component we need is a way of representing a bucket. So, here's our picture of four hash table. What we want is a list, this is going to be the list where each element in the list is a bucket. And what a bucket is, is a list itself where each element in that list is a key and a value. In our case, the key is the word, the value is a list of URLs. So the structure that corresponds to that most closely is this one where we have a list. Each inner list here, so this corresponds to a bucket. And then within the bucket, a word and a list of URLs is one entry. This corresponds to what the entries were in our previous index but now because we want to make it a hash table, we're going to collect them in buckets. So list of those entries and each element in the outer list corresponds to one bucket.

Programming Quiz: Empty Hash Table

So the first thing we're going to do to implement our hash table index, is we have to figure out how to create an empty hash table. With a simple index, this is really easy. To make an empty index, we just have an empty list, there are no elements. So we can initialize the index by just initializing it to an empty list. And as we added elements to the list, we would just add them to the empty list. For the hash table that's not going to work. We need to do something more complicated to start with an empty hash table. And the reason for that is we need to start with all the buckets. So our initial value, for the plain index, was just an empty list.



Our initial value for a hash table needs to be a set of empty buckets. And the reason for that is, we want to be able to do lookups right away, and we want to be able to add elements to our hash table. If we just started with an empty list, well then, the first time we look up a keyword, it would say, that keyword belongs in bucket 27. We don't have a bucket for that. We would need to figure out how to create that bucket. It makes a lot more sense to start by making our empty hashtable be a, list of buckets. Where, initially, all the buckets are empty, they're ready, waiting for keywords to be placed in them. So what we need is code to create that empty hash table. So, I think you know enough to define make_hashtable yourself, so we'll make that a quiz. So your goal is to define a procedure, we'll call it make_hashtable, that takes as input a number, giving the number of buckets in the hash table. And it outputs an empty hash table that has that number of empty buckets.

Answer:

So here's one way to define make_hashtable. We're going to start by initializing a variable i = 0. So we're going to start by creating an empty table and what we want to do is add n buckets, number of buckets to the table. So we're going to use a while loop and we're going to loop while i is less than number of buckets. So each time that I want to go through the loop what we want to do is add on empty bucket to our hash table. So we can do that using Append. That adds a new empty bucket and we need to remember to increase I to make sure that we don't keep looping forever. So we're going to go through this loop buckets number of times each time adding an empty bucket to the table and then we need to return the table at the end.

So let's try that in the Python interpreter. So here's the code just like we read out and we'll print out the result of making a hash table. We'll keep the number of buckets small for printing. For a real use were going to want to have many more than three buckets. And let's run that. And we see what we got is a list with three empty lists as it's elements. So this works okay. It seems like a lot more code than we need. And it is more code than we need. There's a better way to write this, which is to use a four loop. So, the general structure we've seen for four loops. Alright. We've seen a loop that has, has a structure like this, where the collection could be a list. It could be a string. To have a four loop, we need some set of objects that we're looping through. In this case, what we want to do is loop through the numbers from 0 to n buckets minus 1.

So we want to create a list that contains those values. So we, what we would like in order to be able to define a procedure like make hash table, is to have a list, which is the numbers from way to do that, its called range. So range takes two numbers, as inputs. The start and the stop number. And what it outputs is a list of all the numbers from startup to stop minus 1. So this is what ranges outputs, a list of numbers, starting from start. Increasing by 1, until we get to stop minus in is the second parameter in the list. This turns out to be useful because oftentimes when we look through elements, we don't want to include the last element. So that means if we evaluated something like range 0,10, the result would be the list 0,1,2, up to 9. So now that we know about range. We could change our loop here. Instead of having this while loop, we could do the for loop. And we prefer this for two reasons. The first is it's going to make our code shorter. Anytime we can make code shorter, that's usually a good thing. The second is, it saves us from the danger of forgetting to increment the variable. This is a common mistake, and when we forget to increment the variable, the loop's just going to run forever. So if we can write our while loops at for loops, that's usually a good idea. So a better way to define the catch table is to use the for loop. We're no longer going to need the variable i. We still need table and now instead of using a y loop we're going to use a for loop. We're going to leave the variable name blank for a second, we'll figure out what to put there later, and what we're looping through. Is from the range from elements of the list range to buckets. That's going to be the list of numbers from zero to nbuckets, minus 1. And for each one of those we want to append, one new bucket to the table, just like we did before. We don't need to increment i, there's not i variable now. And at the end of the loop, we return the table just as before. For this loop, we didn't actually need a variable here, right? We never used the variable inside. For this index of the four loop, well, we still need something here, so I'm just going to call the variable unused. To make it clear that we have a name there. We don't actually use it in the body of the formula.

So this makes the code a lot smaller. It will work the same way as what we had before. So here's the new code. Several lines shorter than what we had before. Does exactly the same thing. If you were really clever, you might have thought of an even shorter way. To define make hash table, that unfortunately doesn't quite work. So the shorter way would be to guess, that the times operator works on list, the same way it worked on strings. So we could do this by creating empty list, times nbuckets. This seems great, it's only one line, really clear and easy to understand, and it looks like it

almost works. So let's try that in the Python interpreter. And it looks like it worked, we've got a hash table, a list with three empty buckets. There's one big problem with this approach, and I'll show you a hint why it is, and then we'll have a quiz to see if you can figure out why. So now instead of just printing out the result. We're going to assign it to a variable called table, and now we're going to mimic what would happen when we add something to the hashtable. That means we're going to add something to one of the buckets. Let's pick bucket one, and let's assume we're going to add the entry for udacity with one url. And now we can print out what's in that bucket. Looks like everything is okay. What about what's in bucket zero? Now we get the same result. So, think about what went wrong. I'm going to ask a quiz to see if you can understand why this simpler definition of make hash table doesn't actually work correctly.

Quiz: The Hard Way

So the question is, why does this not work? Why does multiplying empty list times n buckets not produce what we need for the empty hash table? And the possible answers are, because it's too easy, and we like doing things the hard way. Because each element in the output refers to the same empty list, or because the store operator means something different for lists than it does for strings.

[ ] Because it is too easy and we like doing things the hard way
[x] Because each element in the output refers to the same empty list
[ ] Because * for lists means something different than it does for strings

Answer:

So the correct answer is the second one. So hopefully, no one picked the first answer. We usually do try to do things the easy way. There is sort of an exception to that in this whole unit. That we're learning ourselves how to define a hash table. Even though Python provides a dictionary type, which I'll talk about at the very end of the unit. That makes things much easier than what we're doing ourselves, but we really want to understand a lot of computer science by building our own hash table, and then we won't actually need to use it because there's a built-in text that works much better. But the answer is the second answer, and here's the reason why.

So what happens when we evaluate this list multiplication, well, we have this empty list, let's draw it like this. And we create a new list, which is three copies of that list. But it's not copies, it's three references to it. So here's the new list. It has three elements. Each one of those elements refers to the same empty list. Then when we did table index 1 append, well, what happened is we follow this reference to whatever table index new element. But because the references at position zero, one, and two, all refer to the same object, that changed all of those values, not just the value of table index 1. And we can see that if we print out the whole table, that the table contains three elements. But they're all the same. And they all refer to the same object. So any change we make to one changes all of them.

Programming Quiz: Finding Buckets

So now that we've created our empty hash table, the next steps are to figure out how to do lookups as well as adds. Both lookups and adds depend on the same first step. We need to find the right bucket, so that's what we'll do next. So remember the idea for a hash table, so we have a list of buckets, each bucket is a list of entries, and each entry is a key, and the list of values. So regardless of whether we want to do lookups and added, and find the value associated with the word, or if we want to do add and add a new value associated with a word. The first thing we always have to do is find the right bucket. So if you find the right bucket if we're doing a lookup, the next thing we're going to do is need to look through all the entries in that bucket to find if there's one that matches the key word. If we're doing add, we also need to start by finding the right bucket. Then we're going to look through and see if that word already exists. If it doesn't exist, we can add a new word and we'll have a new entry, with that word and that value.

So both of those depend on first being able to find the right bucket, so we'll do that once and be able to reuse that code in both lookup and add. So our goal is to define a procedure, and we'll call it hashtable_get_bucket, and it'll take two inputs. So it'll take a hash table, and a keyword, and it will output the bucket where that keyword could occur. We don't know yet whether that keyword's actually in the table. It might be somewhere in that bucket. It might not. But the important thing that hashtable_get_bucket should do, is find the bucket. One function that will be useful for doing that is the hash_string procedure that you defined earlier. And hash_string takes two inputs. It takes a string, which is the keyword, and it took a size, the number of buckets, and it outputs the number which is the index of that bucket. There's a bit of a mismatch here that you'll have to think about to define hashtable_get_bucket. That hash_string, the input is the size, the number of buckets, it's a number. The two inputs to hashtable_get_bucket are just the hash table and a keyword. So you need to also figure out how to get the size of the hash table to pass into hash_string. And if you remember the structure of the hash table, you should be able to figure out how to do that. Remember our hash table is a list of buckets, so the value that we want to pass into the hash_string is the size of that hash table, the number of elements in the list of buckets, that is the hash table. So see if you can define hashtable_get_bucket, and as a hint, you can do the whole definition with a very small amount of code. It should only take one line to do this.

Answer:

Here's the answer; we can define hashtable_get_bucket, and it takes two inputs, the hash table, we'll call that htable, and the key, which is the word we're looking for. And to find the bucket, well, we're going to use hash_string. We're going to pass in the same word, the keyword, that's the input key. The number of buckets is the length of this table, so we're going to call hash_string, passing in the key.

And as the second input, we need the length of the table, that's the number of buckets. So that will get us a number which is the index of the bucket we want. To get that bucket, we need to use that as the index to select that element from htable, and then we want to return the results. So that's all we need to find the bucket. Let's look at that in the Python interpreter. So here's the code we have so far. We have the hash string procedure we defined that maps the key word and a number of buckets to the position where that should occur in the hash table. We have the make_hashtable procedure that creates an empty table with that number of buckets. And now we have the hashtable_get_bucket procedure that takes a hash table and a key. And give this the element of the hash table which is where that key would belong using that hash string function to find the right position.

Programming Quiz: Adding Keywords

So now we know how to find the right bucket. Let's look at how to define the add procedure. So we are going to first define add, we'll make hashtable add, that takes a hash table, a word and a value. And first we'll define a simple version of that, that says we are going to add the new entry to the bucket even if it already exists. So we'll find the right bucket. And we'll add the new entry at the end. So your goal is to define a procedure, we'll call it hash table_add. Takes three inputs: a hash table, a key, which is the word, and a value, and it adds that key to the hash table, making sure to put it in the correct bucket with the associated value that was passed in as the third input.

Answer:

So here's one way we can define hashtable_add. We're going to define the procedure hashtable_add. So the first thing we want to do is find the right bucket. We'll use hashtable_get_bucket we just defined to do that, passing in the table and the key. We'll store that in the variable bucket. The next thing we would need to do is add the new key value pair. And we can do that using append. Append will add a new element to the list. And what we want the list to be, is a new list with two elements, the key and the value. So let's try this in the Python interpreter, and I've done essentially the same thing that we did before, except for we don't really need the variable bucket. We can do the append right away. So i'm getting the bucket and then append the new entry to it. So we'll start, we'll make a hash table. We'll keep the size of the hash table very low. We would never really want a hash table with only three buckets. But to make it easy to look at the results, we'll keep the hash table very small, and let's add to our table a key and value. And for our actual web index the values will be a list of urls. For our test, they could be anything, so lets just use a number, and now we can look at table, and lets run that. And we see we have the table. We have three buckets, and one of the buckets contains an elements. If we look at the bucket that we get from hashtable_get_bucket for the key udacity, we should get the bucket, that contains the word udacity. Lets try that.

That works and we can add some more words to our hash table, more key value pairs to our hash table. So now we've, we have got three entries on our hash table, they all happen to end up in different buckets. That's just lucky in this case. Lets add one more, so we have a bucket that contains more than one entry. And we should print our table after adding one more. And now we have one bucket with one word, one bucket with two entries and one bucket with just one. Now, there is one big problem, with the way we did hashtable_add. Suppose we add the same word again, and let's say now the value of udacity is 27. So now we are making a hash table that has three buckets. We're adding a bunch of entries to it, we've added the entry udacity twice, now when we get the bucket for udacity. What we see is we have a bucket that has the same keyword twice.

If we want our hash table to be a mapping between keys and values, this is going to be a problem. If we have the same keyword twice, when we look up udacity, well, we don't know which answer we should get. Is the value 23 or is the value 27? So, what we want is something a little different from add. What we want is to make sure that every time we add a new keyword to the hash table. If it already exists, instead of having two entries with the same keyword. What we'll do is change the value associated that is associated with that keyword. So we have a problem if we have two entries with the same keyword. We're going to fix that later. Before we fix that, let's define look up. And it will turn out that lookup will actually be helpful in defining a better version of add.

Programming Quiz: Lookup

So, let's think about what we want lookup to do. It's going to take a hash table and a word, its output is going to be the value associated with that keyword. What we want lookup to do is, like add, first thing it has to do is find the right bucket, and we've defined this procedure hashtable_get_bucket to make it easy to do that. Once we've found the bucket, what we want to do for lookup is go through all the entries in that bucket. See if there's one where the key of that entry matches the word that we're passing in.

So, your goals to define a procedure, we call it hashtable_lookup. It takes two inputs, a hash table, and a key, which is the string, and it outputs the value associated with that key. Whenever you're asked to define a procedure, you should think carefully whether the description is clear enough. In this case it's not. There's a situation that this description doesn't cover. We need to think about the case where the key is not actually in the table. What should we do then? There's a lot of different things we could do. We could produce an error. The problem with producing an error is then we've got to deal with it when we call it. We'd like to, instead, produce the value that we can use to represent the case where there's no entry associated with that key. And we'll use the None value for that. So, if the key is not in the table, the result of lookup should be None, which means there's no value associated with that key.

Answer:

So here's one way to define hashtable_lookup. So we're going to take the table and the key. And the first step is to find the bucket, so we'll use the hashtable_get_bucket that we defined earlier for that, and store it in a variable called bucket. So we'll use hashtable_get_bucket to look up in the hash table where that key would occur, and we'll store that in the bucket. But now to do the lookup we need to go through all the entries in that bucket, find one that matches the key. To do the lookup, we need to go through all the entries in the bucket. So we'll use a for loop for that, going through the entries. For each entry, we need to check whether the key part of that entry, matches the key, and remember the bucket was a list of entries. And each entry in the bucket was a keyword in the value. In the case for our web crawler, the value was a list of URLs.

So if the entry matches the key, so the first part of the entry is the keyword, if that matches the key we're looking for, then what we want to do is return the value. So that's the result of entry position one. If it doesn't match, well then we want to keep going, keep trying the next, we're going to keep going through this for loop, checking all the entries in the bucket. When we get to the end, we didn't find it, and what we said in the question was if it's not in the table, what we should do is output none, so now we'll return, the value none. So let's try that in the Python interpreter. So here's the code, we're defining a hashtable_lookup, we find the bucket, we loop through all the entries in the bucket, if we find one where the key matches, we return the value associated with that key, otherwise we return none. When we got to the end of the bucket without finding the entry, we know it couldn't exist anywhere else in the hash table because the only place that keyword could appear is in this bucket. We still have all the same code from before, that makes the hash table, that gets the bucket and then adds to the table. And, we'll test it using some of the code we wrote before.

We're going to add several words to our hash table, and let's try looking up a keyword. We'll look up udacity. And we get the value 23, which is what we expect, that's the value that we associated with udacity. So this is looking pretty good. We've got our hashtable almost working. The one problem is the one we mentioned before, if we try to change the value associated with udacity, and we're going to need to do this for our web index, right? When we add all of the urls to the table, we need to keep adding UR, URLs. We don't want to lose the one we have, but we want to change the value associated with, with that keyword. Now, we do the look up again, and we get the same result. And the reason we got the same result, if we look at the bucket in the hashtable that's associated with the keyword udacity, well what we see is it's a list of two elements. And because of the way we implemented add, the newer one is later in that list, because of the way we define lookup, it's always going to find the first one it matches. It's going through the entries in order. The first one that matches it returns that value. So thats why we get the value 23. So we're really almost done, but we have to make one other change. We want to change add to be more like update. So instead of adding a new entry, we want to update the value that's associated with that entry.

Programming Quiz: Update

So for the last step in implementing our hash table. Your goal is to define a procedure, we'll call it hashtable_update. It takes three inputs the hash table, the key, and the value. And what it should do is update the value associated with that key. So if the key is already in the table, then instead of creating a new entry like were doing before with add. What we want to do is change the value associated with that key, to be the new value that's passed in. If it's not already in the table, then what we want to do is add a new entry, that has that key associated with this value. This is going to be a pretty complicated procedure. I think it's the most complex procedure you've been asked to define so far. But if you think about it carefully and put together all the things we've seen so far, and understand how the hash-table works, I think you'll be able to define it yourself.

Answer:

So updates going to be pretty similar to look up. So let's start by copying that code and seeing how we need to change it. So we have lookup. We're going to change it to be update. Now instead of taking a key as its input, it's going to take a key and a value. But it's not going to return anything, so we're going to get rid of the returns. Right. remember all update is doing is modifying the value of entry. So now we still have what we had for lookup, we're still getting the bucket and we want to do that. We want to make sure that we update the value in the right bucket. We still need to look through the entries in the bucket, to find if one matches. If we find one that matches, well what we did in lookup, was just return it. And update, what we want to do is change the value associated with that key. So we are going to have an assignment that replaces whatever value is there before with a new value. And now instead of returning the value, we want to stop going through the loop, and we actually are done with update, so we can return here. We found the entry, we updated the value. We also need to deal with a case where we didn't find the entry.

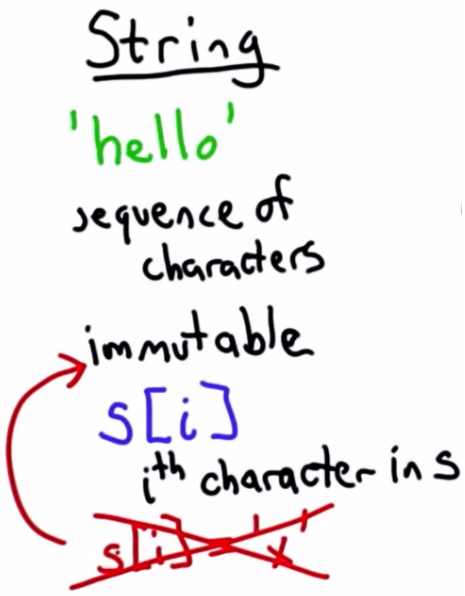
So now we've gone through the loop enough times. When it was a look-up we just returned none. When it's an update what we want to do when the key is not already in the table, is add it. So now we're going to use, append, to add a new entry to bucket, that has the key and the value. So that's how we defined update. There's certainly lots of other ways to do it, and one thing you should be thinking about is, well this is actually very similar to look at, right. We duplicated a lot of code. Maybe there's a way to define update and lookup so we don't have to have two copies of the code, that scans through the bucket to find the right entry. We'll leave that as a homework question for this unit. For now we're going to be happy that we've got correct implementations of both lookup and update. And, let's test them. So, what we did before, we're going to replace the adds with updates, and now, the second time, what happened with the add was we added an entry, but we could never reach that entry because it had the same keyword. Now we're used update, the second time we should be updating that value, and we'll see that the lookup now produces the value 27, for the second lookup. That's good, right, the first time the value was 23. We did the update, we got 27, and we can see that

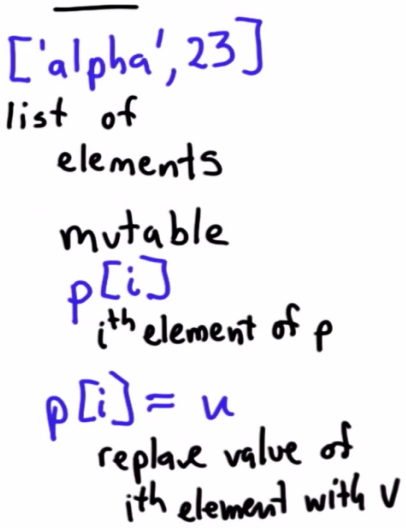
the bucket only contains one entry. So this is great, we finished our implementation of the hash table. We can do updates that will either add new values to the hash table, if they don't already exist, or change the value of ones that exist. And we can do lookups and lookup will know where to look, which bucket to look in to find that key, if it exists.

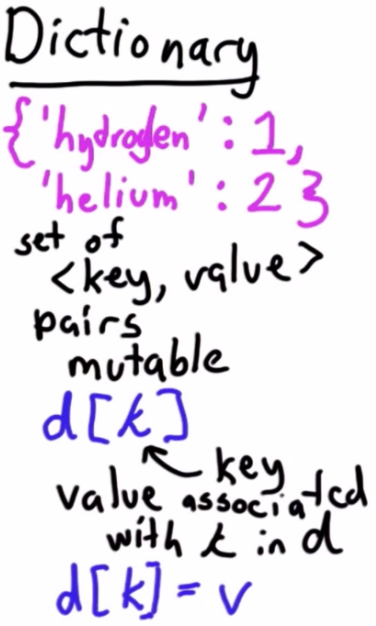
So this has the great property that as the number of keys increase. As long as we increase the numbers of buckets accordingly, the time to do both an update and a lookup is constant. This means the time doesn't increase even as the number of keywords increases, as long as we increase the number of buckets. So, this size of each bucket stays the same size, because the expensive cost of this is going through the elements in the bucket looking for the key that matches.

Dictionaries

So now that we've built it ourselves, I'm going to show you the easy way to do it, which is to use the built in Python type called a dictionary really an implementation of a hash table. It's built into Python. So it's our own hash table. So far, we've seen two complex types in Python. W we've seen the list type. Now we're going to introduce the dictionary t are common between these three. There are other things that are diffe we could have a sequence of characters inside quotes. To create a list, And we could have a sequence of elements inside the square bracket, type, unlike a string where they had to be characters. So, our string wa Our list was a list of any kind of value. So, a list of elements of any valu going to create using the curly bracket. And the entries inside the dictionary are key value pairs. So here I've created a dictionary with two elements. And each element, the key is a string. Here is the string hydrogen and the value associated with that element is a number. The keys in a dictionary can be any immutable value, so they don't need to be strings, they could be numbers. They could be other things. The values can be any value. So what a dictionary is, is a set of key value pairs and the property that a dictionary will give us is like the hash table, that we can look up a key and get a value associated with that key. So one important property of all types is whether they're immutable or mutable. We saw that the string was immutable. That means once we create a string, we can't modify that string. It has the same valu le, we could change value associated with that key in the dictionary. We'll see some exa r indexing and assignments. We saw, with the list, we could do this. the ith element of p with whatever we have here. With the string, w n error. And the reason we can't do it is because strings are immutab characters in a string. With the dictionary we can. And what the me e meaning of update in our hash table. So that's equivalent to updat d with k to be v.





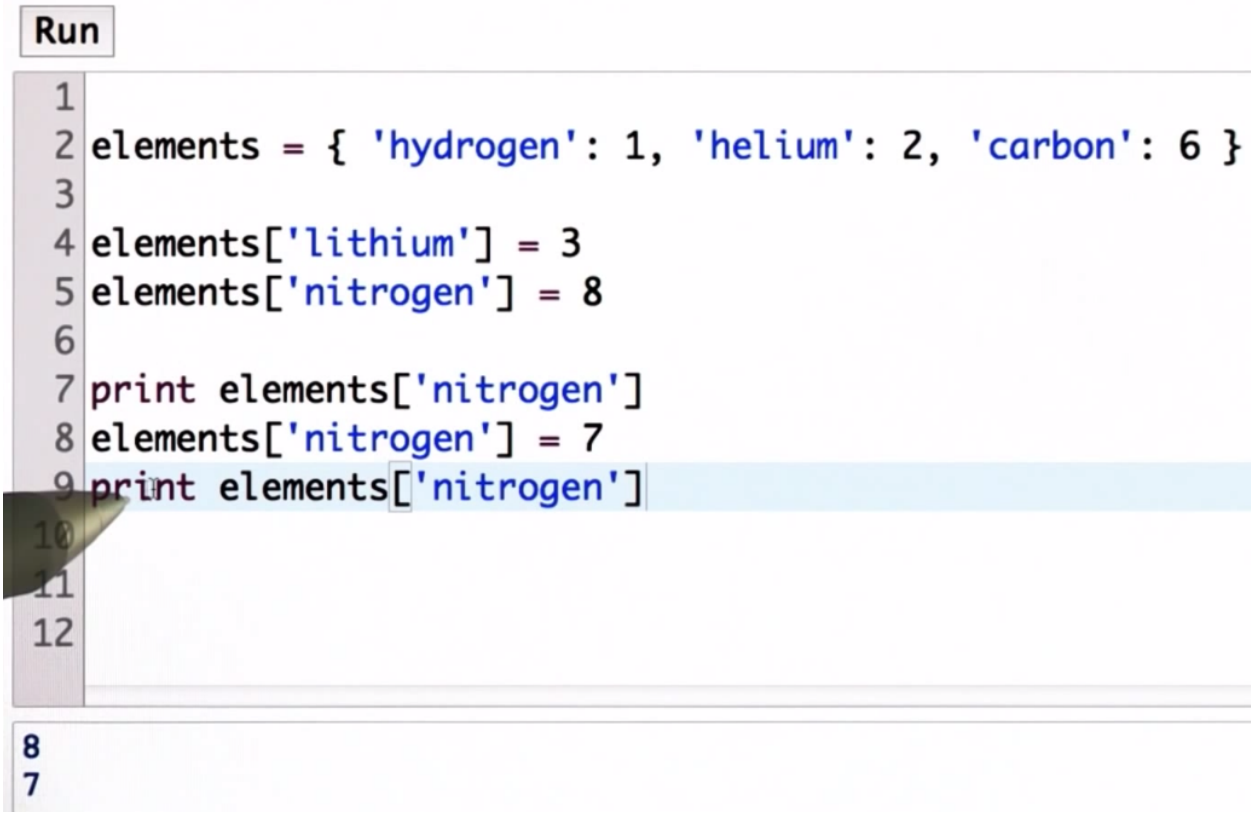


Using Dictionaries

So let's try a few examples. So we'll create a dictionary. We're using the squiggly brackets. On some keyboards that's the shift on the square bracket key. It may be somewhere else on your keyboard. And we're going to use the squiggly bracket to list our elements in the dictionary. So each element in the dictionary is a key value pair, and the entries in the dictionary will be the elements and their atomic numbers. So we have this string hydrogen with its atomic number of 1, and we have carbon with number 6. So we've created the dictionary, and what we see when we print it out is the elements in the dictionary. Now one thing you might notice is they're not in the same order that we put them there. If you think about what we did with hash tables, this isn't so surprising. When we put elements in hash tables, well where they end up in the hash table depends on the key and the hash function. It's not necessarily in the order that we put them in. And with the dictionary, because it's implemented like a hash table, we see the same thing. Unlike a list where the elements are ordered, with a dictionary there is no order to the elements.

So when we print it out, they might appear in a different order from the order that we put them in. We can use the indexing to look up one of the elements. So now when we look up the value associated with hydrogen, we get 1, and if we look up the value associated with carbon, we get 6. What do you think is going to happen if we look up a value that's not in the dictionary? So now we're looking up the value associated with lithium, and what we get is an error. It's called a key error--key error. It says that element is not in the dictionary. This is different from how we defined lookup for our hash table. Right, we defined our lookup to return none when the element was not there. The way the built-in dictionary type works is it gives us an error when it's not there. If we don't want to get errors like that, we can use the in to see if an element is in the dictionary. So the in behaves similarly to in did for lists. We can say lithium in elements. That will evaluate to true if the key is in the dictionary, false otherwise. So now hit evaluate, it's still false.

So I said that dictionaries were mutable. So we can add new elements. And the way to add an element is just to use an assignment. So we'll add lithium giving it a value of 3. Note that it was an error when we used an element like this as a lookup when it's on the left side of an assignment. Well that's an update, so that corresponds to the hash table update where lithium is the key, and 3 is the value. We can add another element. Let's add another element. We'll add nitrogen. And now when we look up nitrogen, we get the value that we assigned to nitrogen, which is 8. It turns out that 8 is not actually the correct value. The atomic number for nitrogen should be 7. So we can modify that value using another assignment. That's again the same as our hash table update. It won't create a new key value because the key nitrogen is already there, but it will update the value, so now the value associated with nitrogen will be 7. So we can see the first print, the value is 8, the second print, the value is now 7.



Programming Quiz: Population

For this quiz, the goal is to see that you understand how to define a dictionary. You should define a dictionary and assign it to the variable population, and it should provide information on the world's largest cities. The key values in your dictionary should be names, the name of each city. And the associated value is the city's population in millions. So that's a number. To get you started, here are the four cities that your dictionary should include. Shanghai's population of 17.8 million, Istanbul with 13.3 million, Karachi, 13.0, and Mumbai with of those four cities, you might want to also include your hometown and any other cities that you're interested in. If you define your dictionary correctly, you should be able to use it like this. Print the population, indexing, looking up the value associated with the key Mumbai, and you should get as output 12.5.

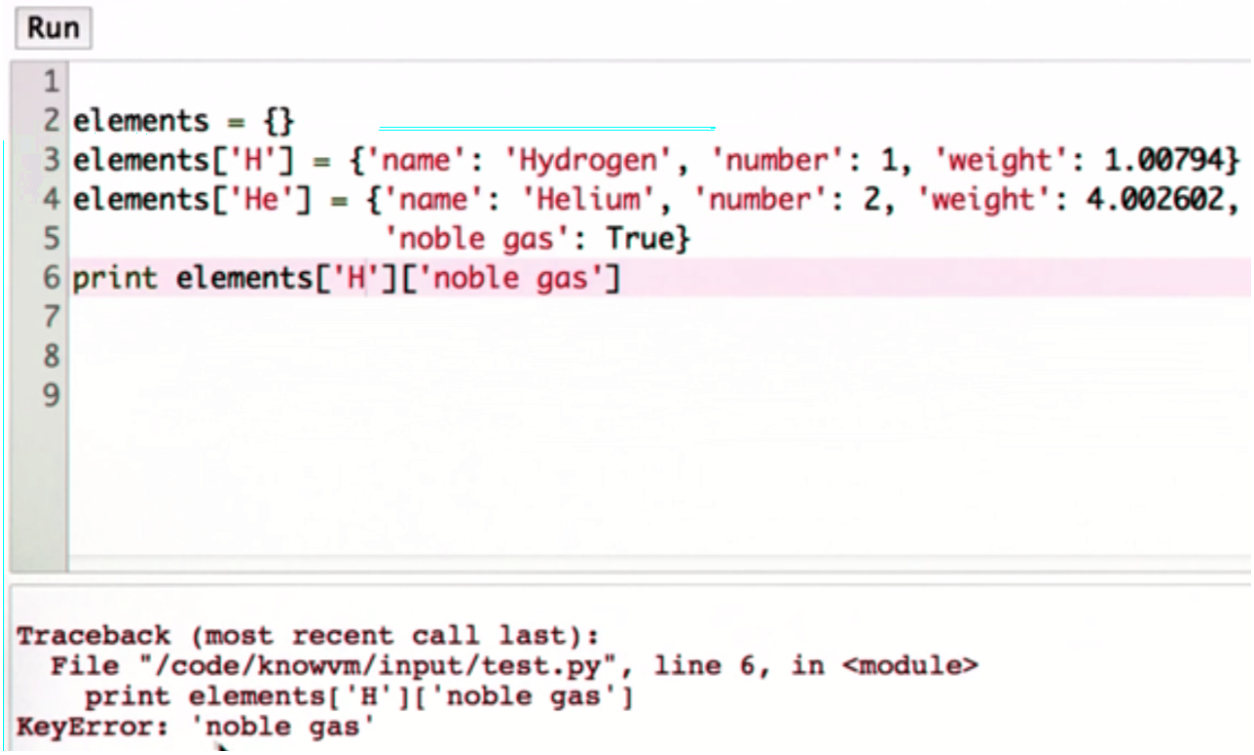
Answer:

So here's one way to solve the quiz; we're going to create a dictionary called Population, and then we'll add some cities to it. We'll add Shanghai, has population 17.8 million. Istanbul has population Mumbai has population 12.5. I'm also going to add my home town, which is Charlottesville, and the population of Charlottesville is pretty small compared to Shanghai. It is 0.043 million people. Only 43,000. So now we've created our dictionary. We've populated it with some entries. We can print those out. So if we look at the value of population of Shanghai, that will get that entry in the dictionary, and

we can see that we get the result 17.8; we can also print out the population of Charlottesville, and we get the result 0.043

A Noble Gas

So here, I've been using just numbers as the values, but the values can actually be anything we want. They can even be other dictionaries. So if you wanted to make more interesting version of elements. So we'll start with an empty dictionary, and we can add a new element. This time we'll use the atomic symbol as the key, and we'll make the value of the element a dictionary that provides some information about it. So we've added an element to our dictionary. The key is the single letter H, and it has as its value a dictionary, that has three entries with the key name number and weight. And values associated with each of those that could be different types. Name is a string, which gives the full name of the element. We'll add another element to our dictionary, and so for helium, we have the same name, number and weight keys, and, so for helium we have an extra entry that says it's a noble gas, and the value of noble gas is true.



So now we can look up the element H, we'll see it's entry. Again note that it's a dictionary, so the order is not the same as the order that we used here. But we can do another lookup. So now we're looking up element H, that gets us the dictionary here. And then we're looking up in that dictionary the value name, and that will give us the name of the element, whose symbol is H. We can change this to look up some other property. Let's look at the weight. And we can change element to look up the value for helium instead of for hydrogen. And now we get the 4.002, which was the weight of helium. If we

look up the noble gas property, we get true for helium. What's going to happen if we look it up for hydrogen? So here we get an error, and we get the same error that we got before when we tried to look lithium up in the elements that didn't include lithium as a key. We are looking for the key noble gas, but it doesn't exist.

Programming Quiz: Modifying The Search Engine

So now I hope you understand how to use dictionaries. And so the next step, we're going to go back to the search engine code from the previous unit, and modify it to use dictionaries instead of the nexus. And this will have the big advantage that now we can do our lookups in constant time as long as we increase the number of buckets with the number of keywords. So to start thinking about this, the question is which of the procedures that we've defined for our search engine will we need to change to make use of the dictionary instead of using the list? So, here's the choices, all the procedures that we've defined so far. So we had a procedure get_all_links, and get_all_links would scan a web page, return a list of all the links in that web page. We had the procedure crawl_web; we have crawl_web, which started with a seed page, and then followed all the links that could be found in that page collecting pages, and for each page, collecting the keywords on that page and adding them to the search index. We have add_page_to_index that would take a page; it was called by crawl_web when it, a new page was found and would add that page to the index by finding all the words on that page, adding them to the index. We had add_to_index, which would take the index, a keyword and the URL where it was found and add that location to the index. And we had lookup, which would take a keyword and give us a list of all the URLs where that keyword would appear. So to answer this, you may need to look at the code, and you'll find it on the website. Think about which ones of these procedures we'll need to change to replace the list index with the dictionary index.

Answer:

So the answers is, we need to change the three of these procedures. We need to change crawl web, we need to change add to index, and we need to change lookup. We don't need to change get all links at all, that we can keep exactly the same as it was. It just returns the list of links, it doesn't depend on the index. We don't need to change add page to index. This is a little more surprising since it depends on index, but because of the way we wrote add page to index, it calls add to index. So it doesn't depend how we actually represent the index. It's going to go through all the words add them to index by calling add to index. So we don't actually have to change that code. We do need to change the other two, so let's start with crawl web and figure out what we need to do to change this, to use a dictionary. And the change is actually going to be really small, right? The index is here, in the old version we initialize index to a list, to the empty list And all we do with index is pass it in to add page to index.

So to change that to use a dictionary, all we need to do is change the square brackets to be curly brackets. So now, instead of starting with an empty list, we're going to start with an empty dictionary. So that's the only change we need to make to crawl_web. The change to add index is going to be a little more complicated. And we can see from the code to crawl_web, what happens with each page was that we call add to index, passing in index, which is now a dictionary. Let's look at add_page_to_index. I claimed that we didn't need to change that. Here's the code to add_page_to_index. And it takes the index. It goes through the words. It adds each word to the index. We can do this just the same whether index was a list or a dictionary. We don't need to change add page to index, we are going to need to change add to index. So we are going to need to change the code to add to index and lets try to figure out how.

So before we had add to index, that takes an index, a keyword and a URL. We'll still take the same parameters but what we had to do when it was a list was go through all the entries in the index, check for each one, if it matches the keyword we're looking for. If we find that it does, then we add the URL. If we get to the end without finding it, then we append a new entry, which is the keyword with a list of URLs containing just the first URL. So let's figure out how to change this to work with the hash table index. So, the great thing about the hash table is we don't need to loop through anything now. We know exactly where it is from the hash table. With the dictionary, the built in in-operation gives us that. So instead of looping, now we can check right away if the keyword is in the index. So what's going to happen if we found the keyword in the index, that means that we can look it up. This will look up in the dictionary the entry that corresponds to the index. That's going to be the list of URLs that we have. And so all we need to do now is append to that entry the new URL. If it's not in the index already, well we need to do something different. What we did before was we added a new element to the index list using append. We don't want to do that now. We want to add a new key value paired to the dictionary. So we're going to do that by using the assignment. And the entry that we're adding is the list containing just this URL. So you can delete everything else here. Add the new entry to the keyword. So this is a lot simpler. We have less code. And it's going to run a lot faster. We don't have to loop through anything. Because of the hash table, we can right away look up whether the keyword is in the index, we can find if it is, what the value is, by using the dictionary lookup, and we can append the newer URL to the list of the URLs associated with that keyword. If it's not found, we can create a new entry, using the dictionary syntax, like this, that contains just that URL. So, now we've got a much simpler way to add to index. I hope you understand this. If you do, you should be able to define lookup yourself.

Programming Quiz: Changing Lookup

For the final quiz for this unit, your goal is to change the lookup procedure to now work with dictionaries. Before we had look up working where index was a list of entries, and we did the look up by looping through the index, that required a lot of work we had to go through each entry, check if the keyword matched, and then return the one that matched, if we got to the end without finding it, we

returned none. Your goal for the quiz, is to modify this code to now work where the index is going to be the dictionary, and we should be able to find the entry much more quickly. We don't need to loop through anything. we can use the property of the dictionary to look up the entry right away. If you understand the add to index code that we just did, I think you can define lookup on your own. The one thing to be careful about, is that you need make sure that when the index does not contain the keyword, when it's not in the dictionary, instead of producing an error, that it returns the value none.

Answer:

Here's one way to define look up. Instead of needing the loop, we're just going to need to check to if the keyword is in the index, if it is, we can return the value associated with that keyword using the dictionary look up like this. If it isn't, we return the value none.

Coming Up Next

Congratulations, you have completed unit 5. You have a search engine that can respond to queries quickly, no matter how large the index gets. This problem of analyzing the cost of algorithms and designing data structures that are more efficient, is one of the core ideas in all of computer science. It's something we go into a lot more depth on in later courses that I hope you will take after finishing CS101, including the Algorithms and Theory of Computation course. The main thing we have left to do for our search engine, is to find a way to get the best page for a given query, instead of all pages. This is the main thing that made Google so successful - figuring out a better way to identify the pages searchers really care about. Thats what we are going to look at in unit 6. Hope to see you back soon!