

CS101 - Unit 5: How Programs Run - Making Things Fast

Contents

- 1 [Introduction](#)
- 2 [Making Things Fast](#)
- 3 [What is Cost?](#)
 - 3.1 [Quiz: Measuring Speed](#)
- 4 [Stopwatch](#)
- 5 [Spin Loop](#)
- 6 [Predicting Run Time](#)
 - 6.1 [Quiz: Predicting Run Time](#)
- 7 [Make Big Index](#)
 - 7.1 [Quiz: Index Size Vs. Time](#)
 - 7.2 [Quiz: Lookup Time](#)
- 8 [Worst Case](#)
 - 8.1 [Quiz: Worst Case](#)
 - 8.2 [Quiz: Fast Enough](#)
- 9 [Making Lookup Faster](#)
 - 9.1 [Quiz: Hash Table](#)
- 10 [Hash Function](#)
- 11 [Modulus Operator](#)
 - 11.1 [Quiz: Modulus Quiz](#)
 - 11.2 [Quiz: Equivalent Expressions](#)
- 12 [Bad Hash](#)
 - 12.1 [Quiz: Bad Hash](#)
- 13 [Better Hash Function](#)
 - 13.1 [Quiz: Better Hash Functions](#)
- 14 [Testing Hash Functions](#)
 - 14.1 [Quiz: Keywords and Buckets](#)
- 15 [Implementing Hash Tables](#)
 - 15.1 [Quiz: Implementing Hash Tables](#)
- 16 [Empty Hash Table](#)
 - 16.1 [Quiz: Empty Hash Table](#)
 - 16.2 [Quiz: The Hard Way](#)
- 17 [Finding Buckets](#)
 - 17.1 [Quiz: Finding Buckets](#)
 - 17.2 [Quiz: Adding Keywords](#)
 - 17.3 [Quiz: Lookup](#)
 - 17.4 [Quiz: Update](#)
- 18 [Operations on Hash Table](#)

- 19 [Dictionaries](#)
- 20 [Using Dictionaries](#)
 - 20.1 [Quiz: Population](#)
- 21 [A Noble Gas](#)
- 22 [Modifying the Search Engine](#)
 - 22.1 [Quiz: Modifying the Search Engine](#)
 - 22.2 [Quiz: Changing Lookup](#)
- 23 [Changing Lookup](#)

Introduction

In the last unit you built a search index that could respond to queries by going through each entry one at a time. The search index checked to see if the keyword matched the word you were looking for and then responding with a result.

However, with a large index and lots of queries, this method will be too slow. A typical search engine should respond in under a second and often much faster.

In this unit you will learn how to make your search index much faster.

Making Things Fast

The main goal for this unit is to develop an understanding of the cost of running programs. So far, you haven't worried about this and have been happy to write code that gives the correct result. Once you start to make programs bigger, make them do more things or run on larger inputs, you have to start thinking about the cost of running them. This question of what it costs to evaluate an execution is a very important, and it is a fundamental problem in computer science. Some people spend their whole careers working on this. It's called **algorithm analysis**.

You may not be aware of this but you've already written many algorithms. An **algorithm** is a procedure that always finishes and produces the correct result. A **procedure** is a well defined sequence of steps that it can be executed mechanically. We're mostly interested in procedures which can be executed by a computer, but the important part about what makes it a procedure is that the steps are precisely defined and require no thought to execute.

To be an algorithm, it has to always finish. You've already seen that it isn't an easy problem to determine if an algorithm always finishes. It isn't possible to answer that question in general, but it can be

answered for many specific programs.

So, once you have an algorithm, you have well a defined sequence of steps that will always finish and produce the right results. This means you can reason about its cost.

What is Cost?

The way computer scientists think about cost is quite different from how most people think about cost.

When you think about the cost of things, you know specific things such as the red car costs \$25000 and the green car \$10000 - the red car costs more than the green car. You just have to compare those costs. This is thinking in terms of very specific things with specific costs. It's different with algorithms. We don't usually have a specific execution in mind. The cost depends on the inputs.

Suppose algorithms Algo 1 and Algo 2 both solve the same problem.

- Inputs $\hat{+}$ Algo 1 $\hat{+}$ Output
- Inputs $\hat{+}$ Algo 2 $\hat{+}$ Output

You can't put a fixed price on them like with the cars. For some inputs, it might be the case that Algo 1 is cheaper than Algo 2, but for others, Algo 2 might be cheaper. You don't want to have to work this out for every input because then you might as well run it for every input. You want to be able to predict the cost for every input without having to run every input.

The primary way that cost is talked about in computer science is in terms of the size of the input. Usually, the size of the input is the main factor that determines the speed of the algorithm, that is, *the cost in computing is measured in terms of how the time increases as the size of the input increases*. Sometimes, other properties of the input matter, which will be mentioned later. Ultimately, cost always comes down to money. What costs money when algorithms are executed?

- The time it takes to finish - if it finishes more quickly, you'll spend less time on it. You can rent computers by the time it takes to execute. There are various cloud computing services where you pay for a certain sized processor for the time you use it. It's just a few cents per hour. Time really is money and, although we don't need to turn the costs into money because we might not know the exact computing costs, understanding the time to execute will give a good sense of the cost.

- Memory - if a certain amount of memory is needed to execute an algorithm then you have an indication of the size and cost of computer required to run the program.

NOTE: So we want to know how cost(time,memory) depends on the input.

In summary, cost is talked about in terms of time and memory rather than money; although the actual implementation of these do convert to actual monetary costs. Time is often the most important aspect of cost, but memory is also another consideration.

Quiz: Measuring Speed

Why do computer scientists focus on measuring how time scales with input size, instead of absolute time? (Check all correct answers.)

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

Throughout the entire history of computing, it's been the case that the computer you can buy in a year for the same amount of money will be faster than the computer you can buy today.

[Answer](#)

Stopwatch

In this section, you'll see how you can check how long it takes for a piece of code to run. You may have seen people on the forums talking about bench marking code.

The procedure, **time_execution**, below, is a way to evaluate how long it takes for some **code** to execute. You could try to do this with a stopwatch but to be accurate, you'd have to run programs for a very long time. It's more accurate to use the built-in **time.clock** in the **time** library. An explanation of what is going on follows the code:

```
import time #this is a Python library
```

```
def time_execution(code):
    start = time.clock() # start the clock
    result = eval(code) # evaluate any string as if it is a P
    run_time = time.clock() - start # find difference in start
    return result, run_time # return the result of the code and
```

The clock is started. The time it reads when it starts is somewhat arbitrary, but it doesn't matter because you're only interested in the time difference between it starting and stopping and not an absolute start and end time. After the clock is started, the **code** you want to evaluate is executed. This is done by the rather exciting method **eval('<string>')**. It allows you to run any code input as a string! You put in a string and it runs it as Python code. For example, **eval('1 + 1')** runs the code **1 + 1**. After the code is executed, the clock is stopped and the difference between the start and stop times is calculated and stored as **run_time**. Finally, the procedure returns the result of the **code** in **eval** and its running time.

You can run the timing through the web interpreter, but it won't be accurate and there is a limit on how long your code is allowed to run there. If you have Python installed on your computer, you'll be able to run it on that. If you don't, then that's ok. Instructions will be available under the unit's Supplementary Information. You don't need to run it for yourself, but you should at least see it illustrated. The following outputs are all run in a Mac shell on Dave's desktop.

Recall that the input **'1 + 1'** to **time_execution**, shown in green in the image below, is sent as input to **eval**, which runs **1 + 1** as Python code.

The run time is written in scientific notation:

8.300000000005525e-05 which is sometimes also written as

8.300000000005525 x 10⁻⁵, or in Python code

8.300000000005525 * 10-5**. In decimal terms, this can be written as **0.0000830000000005525**, or rounded to **0.000083**.

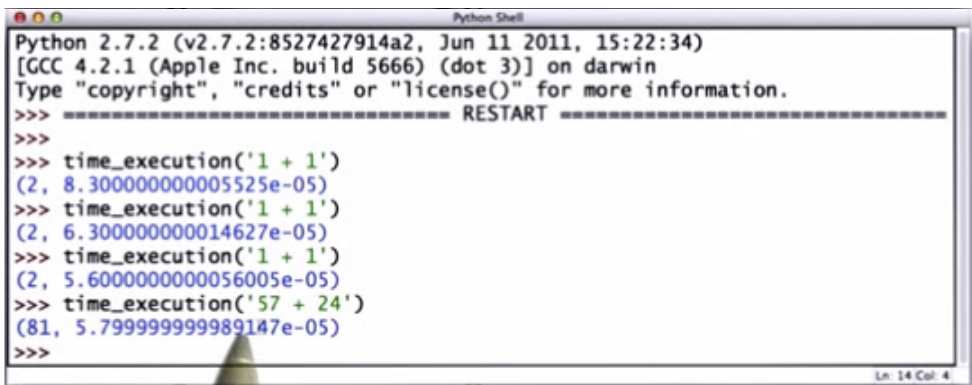
You can see where this comes from by looking at the **-5** after the **e**. It tells you to move the decimal point 5 steps to the left like this:



```
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 =====
>>>
>>> time_execution('1 + 1')
```

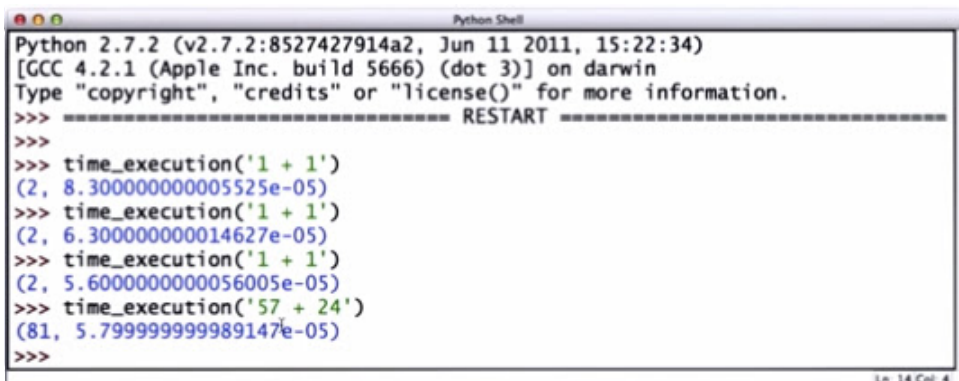
The units are seconds, so this is only a fraction of a millisecond, that is, about 0.08 ms.

Trying the same instruction over and over, the result varies because other things are going on in the machine at the same time, but it's around the same value.



```
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 =====
>>>
>>> time_execution('1 + 1')
(2, 8.30000000005525e-05)
>>> time_execution('1 + 1')
(2, 6.300000000014627e-05)
>>> time_execution('1 + 1')
(2, 5.600000000056005e-05)
>>> time_execution('57 + 24')
(81, 5.799999999989147e-05)
>>>
```

Instead, try larger numbers, the time is still very very small.



```
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 =====
>>>
>>> time_execution('1 + 1')
(2, 8.30000000005525e-05)
>>> time_execution('1 + 1')
(2, 6.300000000014627e-05)
>>> time_execution('1 + 1')
(2, 5.600000000056005e-05)
>>> time_execution('57 + 24')
(81, 5.799999999989147e-05)
>>>
```

The actual processing time is even lower, because, for instance, starting and stopping the clock.

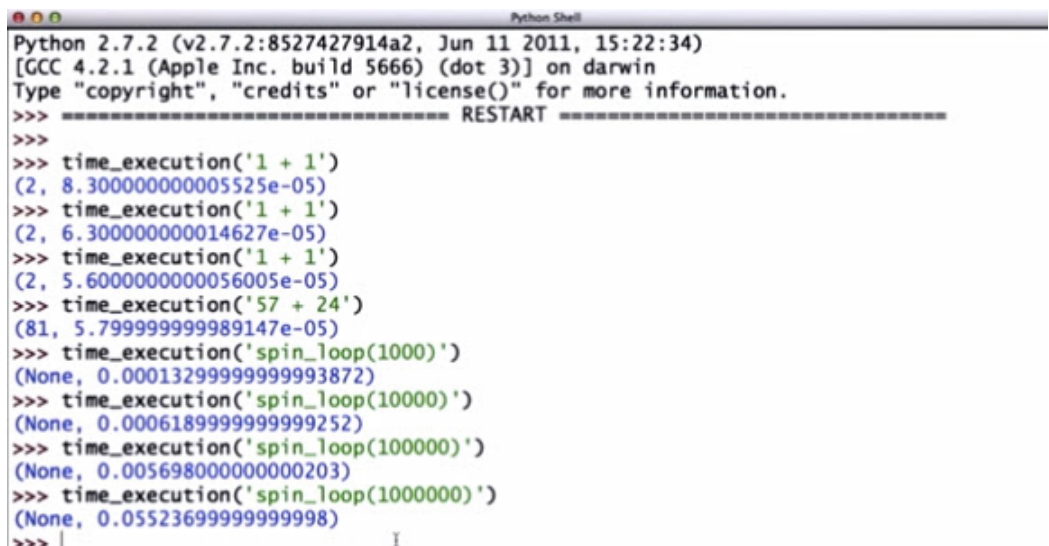
This doesn't tell you very much for short, fast executions, so next you'll see some longer ones.

Spin Loop

In order to get a better idea of how timing works, define the procedure **spin_loop**:

```
def spin_loop(n):
    i = 0
    while i < n:
        i = i + 1
```

This code will run for longer, and by picking the value *n* you can go through and loop any number of times. The image below shows **spin_loop** running 1000, 10000, 100000 and 1000000 times, returning a result that is the time it takes to run the loop that number of times.



```
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 =====
>>>
>>> time_execution('1 + 1')
(2, 8.300000000005525e-05)
>>> time_execution('1 + 1')
(2, 6.300000000014627e-05)
>>> time_execution('1 + 1')
(2, 5.6000000000056005e-05)
>>> time_execution('57 + 24')
(81, 5.799999999989147e-05)
>>> time_execution('spin_loop(1000)')
(None, 0.00013299999999993872)
>>> time_execution('spin_loop(10000)')
(None, 0.0006189999999999252)
>>> time_execution('spin_loop(100000)')
(None, 0.005698000000000203)
>>> time_execution('spin_loop(1000000)')
(None, 0.05523699999999998)
>>> |
```

It is important to notice that the time changes depending on the input - when you increase the input, the time (or the output) also increases accordingly.

Predicting Run Time

First you'll see the time taken for running some code, and then there will be a quiz. This will show if you understand execution time well enough to make some predictions.

The code used to measure the time is the **time_execution**

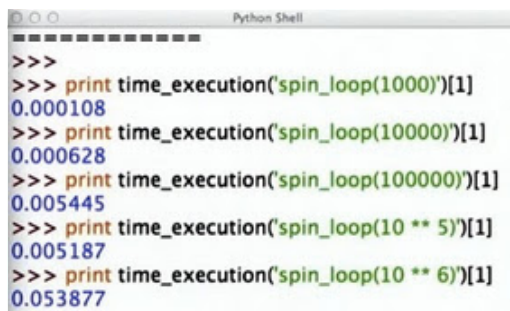
procedure from before which evaluates the time taken for the **code** passed in, and then a procedure **spin_loop** which just adds one to a variable as it loops through the numbers up to **n**.

```
import time

def time_execution(code):
    start = time.clock()
    result = eval(code) # evaluate any string as if it is a p
    run_time = time.clock() - start
    return result, run_time

def spin_loop(n):
    i = 0
    while i < n:
        i = i + 1
```

The results for execution times, in seconds, for **spin_loop** are given below. Note the **[1]** is there to just print out the running time rather than the result of evaluating **spin_loop**. The first result is for running the loop 1000 times, then 10000 followed by 100000. The next is a repeat of the 100000 iterations (runs) through the loop, but written in Python's power notation for 10^5 . The final time is the execution time for running through the loop 10^6 , which is one million times. All the execution times are given in seconds.



```
Python Shell
>>>
>>> print time_execution('spin_loop(1000)')[1]
0.000108
>>> print time_execution('spin_loop(10000)')[1]
0.000628
>>> print time_execution('spin_loop(100000)')[1]
0.005445
>>> print time_execution('spin_loop(10 ** 5)')[1]
0.005187
>>> print time_execution('spin_loop(10 ** 6)')[1]
0.053877
```

Quiz: Predicting Run Time

Given the execution times above, what is the expected execution time for **spin_loop(10**9)** (one billion) in seconds? You won't be able to guess the exact time, but the grader is looking for a multiple of 5.

[Answer](#)

Make Big Index

The examples you've seen so far have shown the time taken to run short procedures. Next you'll see some test on longer code - the index code, which is important to the overall look up time of your search engine. In order to test this code, you'll first need to build a big index. You could do this by hand but it would take a very long time! The code to do it is as follows.

```
def make_big_index(size):
    index = []
    letters = ['a','a','a','a','a','a','a','a']
    while len(index) < size:
        word = make_string(letters)
        add_to_index(index, word, 'fake')
        for i in range(len(letters) - 1, 0, -1):
            if letters[i] < 'z':
                letters[i] = chr(ord(letters[i])+ 1)
                break
            else:
                letters[i] = 'a'
    return index
```

First an empty index called **index** is created, and a list, **letters**, of eight letter a's. Next, there is a **while** loop that adds an entry to the index each time it iterates. (Iterates means going through the loop - one iteration is one pass through the loop.) The while loop continues to add to the index until it is of the required length, size. The details as to what happens in loop are as follows.

```
word = make_string(letters)
```

The procedure **make_string** takes as its input **letters** and returns a single string of the entries contained in the list. This means that, for example, **['a','a','a','a','a','a','a','a']** becomes **'aaaaaaaa'**. The code for this is:

```
def make_string(p):
    s=""
    for e in p: # for each element in the list p
        s = s + e # add it to string s
    return s

add_to_index(index, word, 'fake')
```

This is the **add_to_index** code from before, and it adds an entry which looks like this:

```
['aaaaaaaa', ['fake']] to index.
```

```
for i in range(len(letters) - 1, 0, -1):
```

Although you've seen **range** before, here it is used differently. If **len(letters)** is 8, then **range(len(letters) - 1, 0, -1)** becomes **range(7, 0, -1)**, which is a list which counts down from 7 to one greater than 0 in steps of -1, that is, *[7, 6, 5, 4, 3, 2, 1]*. So, the for loop runs through the values of the list counting backwards from **len(letters)-1** down to 1.

Discussion on Range

The **for** loop starts from the last letter in **letters**, and checks if it is a 'z'. If it is, it changes it to 'a', and goes back to the top of the loop for the next iteration, which will check one position closer to the beginning of the loop. It continues until it finds a letter which is not a 'z'.

Once it finds a letter which isn't a 'z', the line **letters[i] = chr(ord(letters[i]) + 1)** changes it to one letter later in the alphabet, 'a' to 'b', 'b' to 'c' and so on. (Precisely what **chr** and **ord** do will be explained later.) After a letter has been changed, the **for**-loop stops and the code returns to the top of the **while**-loop.

During the first iteration of the **while**-loop, the **for**-loop it will change from: **['a','a','a','a','a','a','a','a']** to **['a','a','a','a','a','a','b','a']** and then break. The second time through, it will change from: **['a','a','a','a','a','a','b','a']** to **['a','a','a','a','a','a','c','a']** and break, and so on.

When it gets to: **['a','a','a','a','a','a','a','z']**, it will change it first to: **['a','a','a','a','a','a','a','a']**.

Then it will look at the last but one position, as it goes through the list backwards. It will change the 'a' in that position to 'b' which changes the list to: **['a','a','a','a','a','a','b','a']**, after which it breaks.

Finally, when the **while** loop has finished, the index of length **size** is returned.

The complete code to try this is reproduced below, so you can try it for yourself if you'd like.

```
def add_to_index(index, keyword, url):
    for entry in index:
        if entry[0] == keyword:
```

```

        entry[1].append(url)
        return
    index.append([keyword, url])

def make_string(p):
    s=""
    for e in p:
        s = s + e
    return s

def make_big_index(size):
    index = []
    letters = ['a','a','a','a','a','a','a','a']
    while len(index) < size:
        word = make_string(letters)
        add_to_index(index, word, 'fake')
        for i in range(len(letters) - 1, 0, -1):
            if letters[i] < 'z':
                letters[i] = chr(ord(letters[i])+ 1)
                break
            else:
                letters[i] = 'a'
    return index

```

To see what it does, you can look at some small values.

```

print make_big_index(3) # index with 3 keywords
['aaaaaaaa', ['fake'], ['aaaaaaab', ['fake']], ['aaaaaaac', ['f

print make_big_index(100) # index with 100 keywords
['aaaaaaaa', ['fake'], ['aaaaaaab', ['fake']], ['aaaaaaac', ['f
... <snip> ...
['aaaaaaadm', ['fake']], ['aaaaaadn', ['fake']], ['aaaaaado',

```

As you can see from the end of the list, the second to last letter has been changed as well as the last index.

To test the index, some larger indexes are needed. You'll see the results of tests run on one of length 10 000, and another of length 100 000. It takes over 5 minutes to construct the index of length 100 000. This doesn't matter as it is the lookup time that is important for your search engine.

To construct the smaller of the two indexes, you can use the line:

```
index10000 = make_big_index(10000)
```

The time to run this **lookup** on **index10000** is checked several times, which you can see below. Note that the time varies, but is

around the same value.

```
>>> index10000 = make_big_index(10000)
>>> time_execution('lookup(index10000, "udacity")')
(None, 0.0008559999999997459)
>>> index100000 = make_big_index(100000)
>>> print index100000[99999]
['aaaafryd', ['fake']]
>>> print index100000[-1]
['aaaafryd', ['fake']]
>>> time_execution('lookup(index10000, "udacity")')
(None, 0.0009680000000000302)
>>> time_execution('lookup(index10000, "udacity")')
(None, 0.00090599999999863066)
>>> time_execution('lookup(index10000, "udacity")')
```

After constructing the larger index of length 100 000 using:

```
index1000000 = make_big_index(100000)
```

you can see the element at the last position using

index1000000[99999], or the equivalent **index1000000[-1]**, just like with strings.

```
print index1000000[99999]
['aaaafryd', ['fake']]
index1000000[-1]
['aaaafryd', ['fake']]
```

The timings for the longer index of length 100 000 are as follows:

```
>>> time_execution('lookup(index100000, "udacity")')
(None, 0.008590000000002652)
>>> time_execution('lookup(index100000, "udacity")')
(None, 0.008517999999998093)
>>> |
```

If you compare these with the times for the index of length 10 000, you'll see that it takes approximately 10 times longer to do a lookup in the larger index than in the shorter index.

Timings vary for many reasons. One reason is that lots of other things run on the computer, which means that the program does not have total control over the processor. Another reason is that where things are in memory can take a longer or shorter time to retrieve. What matters is that the run time is roughly the same each time the test is run, and that it depends on the input size.

Quiz: Index Size Vs. Time

What is the largest size index that can do lookups in about one second?

1. 200 000 keywords
2. 1 000 000 keywords
3. 10 000 000 keywords
4. 100 000 000 keywords
5. 1 000 000 000 keywords

Sample timings:

```
>>> time_execution('lookup(index10000, "udacity")')
(None, 0.000968000000000302)
```

```
>>> time_execution('lookup(index10000, "udacity")')
(None, 0.000905999999863066)
```

```
>>> time_execution('lookup(index100000, "udacity")')
(None, 0.008590000000002652)
```

```
>>> time_execution('lookup(index100000, "udacity")')
(None, 0.00851799999998093)
```

[Answer](#)

Quiz: Lookup Time

Predict the lookup time for a particular keyword in an index created as given below. Be careful, this is a bit of a trick question.

What is the expected time to execute:

```
lookup(index10M, 'aaaaaaaa')
```

where index10M is an index created by

```
index10M = make_big_index(10000000)
```

1. 0.0 s
2. 0.1 s
3. 1.0 s
4. 10 s

[Answer](#)

Worst Case

Usually, when analysing programs it's the worst-case execution time that is important. The **worst-case** execution time is the time it takes for the case where the input for a given size takes the longest to run. For **lookup**, it's when the **keyword** is the last entry in the index, or not in the index at all. Looking at the code will give you a better understanding of why the time scales as it does, and the worst-case and the average case running times.

This is the code from the last unit.

```
def add_to_index(index, keyword, url):
    for entry in index:
        if entry[0] == keyword:
            entry[1].append(url)
            return
    # not found, add new keyword to index
    index.append([keyword, [url]])

def lookup(index, keyword):
    for entry in index:
        if entry[0] == keyword:
            return entry[1]
    return None
```

What **lookup** does is go through the list. For each entry, it checks if it is equal to the keyword. Below is the structure of the index.

number of times through the loop depends on len(index)

```
Run
1 def add_to_index(index, keyword, url):
2     for entry in index:
3         if entry[0] == keyword:
4             entry[1].append(url)
5             return
6     # not found, add new keyword to index
7     index.append([keyword, [url]])
8
9 def lookup(index, keyword):
10    for entry in index:
11        if entry[0] == keyword:
12            return entry[1]
13    return None
14
```

The number of iterations through the loop depends on the index. The length of the index is the maximum number of times the code goes through the loop. If the keyword is found early, the loop finishes sooner.

The other thing that is relevant is how **add_to_index** constructs

the list. It loops through all the entries to see if the **keyword** exists, and if it doesn't, it adds the new entry to the end of the list. The first addition is added to the beginning of the **index** and the last to the end. That is why **'aaaaaaa'** is the first entry in the **index**.

Quiz: Worst Case

Which keyword will have the worst case running time for a given index? (Choose one or more answers.)

1. lookup(index, first word added)
2. lookup(index, word that is not in index)
3. lookup(index, last word added)

Quiz: Fast Enough

This is a fairly subjective question.

Is our lookup fast enough?

1. Yes.
2. It depends on how many keywords there are.
3. It depends on how many urls there are.
4. It depends on how many lookups there are.
5. No.

[Answer](#)

Making Lookup Faster

How can we make **lookup** faster? Why is it so slow?

It's slow because it has to go through the whole of the **for** loop

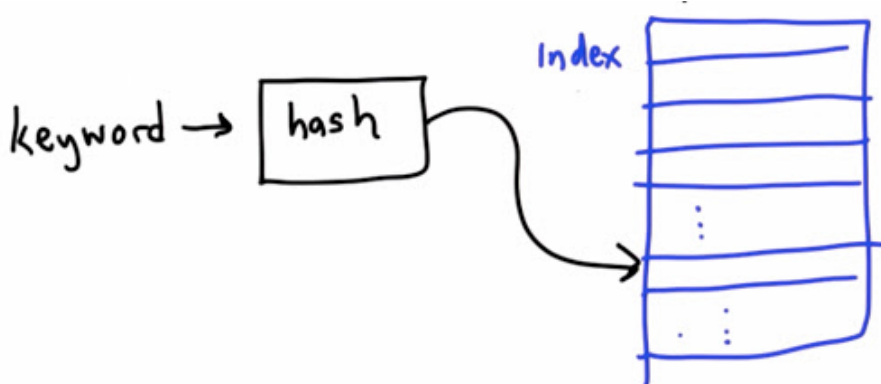
```
for entry in index:  
    if ...
```

to find out if a keyword isn't there. This isn't how you use an index in real life. When you pick up a book, and look in the index, you don't start at A and work your way all the way through to Z to know if an entry isn't there. You go directly to where it should be. You can jump around the index because it is in sorted order. You know where the entry belongs because it is in alphabetic order. If it isn't where it belongs, it isn't in the index anywhere.

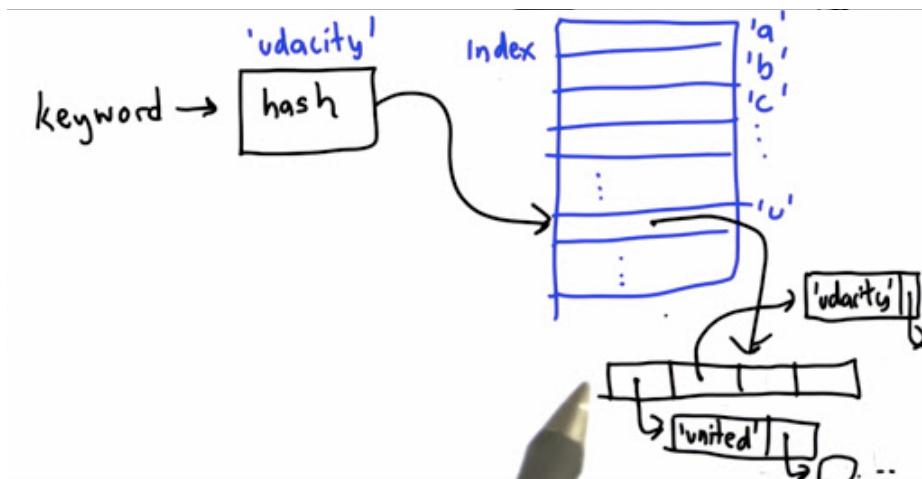
You could do something similar with the index in your code. If you had a sorted index, you could go directly to the entry you wanted.

Sorting is an interesting problem, but it won't be covered in this course. Instead of having to keep the index in a specific order, you'll learn another way to know where to look for the entry you are interested in. This method will be a function, called a **hash function**. Given a keyword, the hash function will tell you where to look in the index. It will map the keyword to a number (this means it takes in a keyword and outputs a number) which is the position in the index where you should look for the keyword. This means you don't have to start at the beginning and look all the way through the index to find the keyword you are looking for.

NOTE: Hash function tells you where to look(which bucket to look in) to find the entry you are looking for.



There are lots of different ways to do this. A simple way would be to base it on the first letter in each keyword. It would be like the way an index in a book works. Each entry in the index will correspond to a letter and will contain all the keywords beginning with that letter.



This isn't the best way to do it. It will reduce the time it takes to

search through all the keywords as you'll only need to search through the keywords beginning with that letter, but it won't speed it up that much. The best it could do is to speed up the look up by a factor of 26, as there are 26 buckets, so each list would be 26 times smaller if all the buckets were the same size. It wouldn't be that good for English, since there are many more words beginning with S or T than there are beginning with X or Q, so the buckets are very different sizes. If you have millions of keywords, it will be faster, but not fast enough. There are two problems to fix:

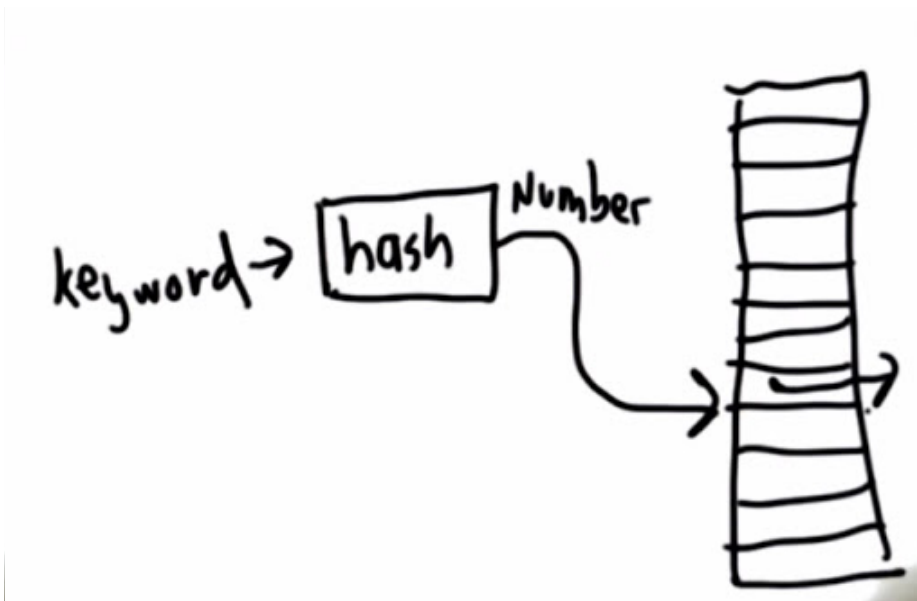
- Make a function depending on the whole word
- Make the function distribute the keywords evenly between the buckets.

The table described is called a **hash table**. It's a very useful data structure. In fact, it is so useful that it's built into Python. There's a Python type (types are things like index and string) called the **dictionary**. It provides this functionality. At the end of this unit, you'll modify your search engine code to use the Python dictionary. However, before you learn about that, you'll learn to implement it yourself to make sure you understand how a hash table works.

My Notes: A Hash Table is a table with numbered bucket slots. Using a hash functions, an entry can be placed in a certain bucket by using the entry's keyword. The hash function maps a keyword to a number using the keyword's characters. That number is the bucket number in which that entry belongs in. It makes look up a lot faster....

Quiz: Hash Table

Suppose we have b buckets and k keywords ($k > b$). Which of these properties should the hash function have? Recall that a hash function is a function which takes in a keyword and produces a number. That number gives the position in the table of the bucket where that input should be.

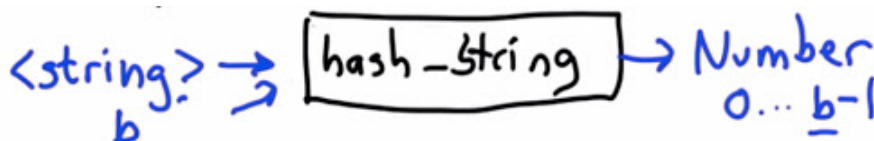


1. output a unique number between 0 and k-1.
2. output a unique number between 0 and b-1.
3. map approximately k/b keywords to bucket 0.
4. map approximately k/b keywords to bucket b-1.
5. map more keywords to bucket 0 than to bucket 1.

[Answer](#)

Hash Function

Next, define a hash function which will satisfy those properties. It will be called **hash_string**. It takes as its inputs a string and the number of buckets, b, and outputs a number between 0 and b-1.



What you will need, which has not been discussed yet, is a way to turn a string into a number. You may recall that there were two operators in the code to generate the big index, **make_big_index**, that were not explained. These were **ord** (for ordinal) and **chr** (for character). The operator **ord** turns a one-letter string into a number, and **chr** turns a number into a one-letter string.

```
ord(<one-letter string>) â†’ Number
chr(<Number>) â†’ <one-letter string>
```

Examples:

```
print ord('a')
97
print ord('A')
65
print ord('B')
66
print ord('b')
98
```

Note that upper and lower case letters are mapped to different letters. Also note that the number for '**B**' is higher than for '**A**', and for '**b**' it is higher than for '**a**'.

A property of **ord** and **chr** is that they are **inverses**. ((One function is the 'reverse' of the other)) This means that if you input a single letter string to **ord** and then input the answer to **chr**, you get back the original single letter. Similarly, if you input a number to **chr** and then input the answer to that into **ord**, you get back the original number (as long as that number is within a certain range.) This means that for any particular character, which we'll call alpha **a**, **chr(ord(a))** is **a**. For example, in the code:

```
print chr(ord(u))
u
print ord(chr(117))
117
```

You can see below what happens if you enter a number which is too large.

```
print ord(chr(123456))
Traceback (most recent call last):
  File "C:\Users\Sarah\Documents\courses\python\testing.py", line 1, in <module>
    print ord(chr(123456))
ValueError: chr() arg not in range(256)
```

From the last line of the error message, [ValueError: chr\(\) arg not in range\(256\)](#), you can see that **chr()** requires its input to be in the list of integers given by **range(256)**, which is a list of all the integers from 0 to 255 inclusive.

The numbers given by **ord** are based on [ASCII](#) character encoding. What these numbers are doesn't matter for the purpose of making a hash table. All that is important is that they are different for each letter. You'll be able to use **ord** to turn characters into numbers. The

limitation of `ord` is that it can only take one letter as input. If you try to input more, you'll get an error message. `{#!highlight python print ord('udacity')}` Traceback (most recent call last):

- File "C:\Users\Sarah\Documents\courses\python\testing.py", line 2, in <module>
 - `print ord('udacity')`

TypeError: `ord()` expected a character, but string of length 7 found
 }}} The last line tells you that you should just input a character, i.e. a single letter string, but instead you've input a string of length 7.

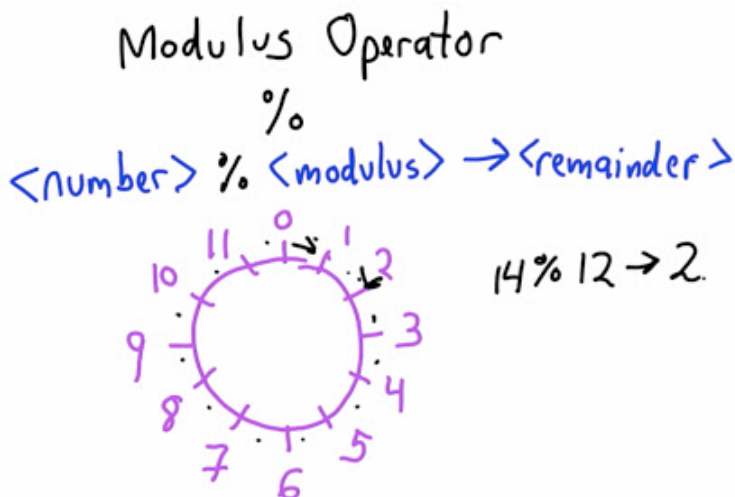
You now have a way to convert single character string to numbers which is **`ord`**. Next, you'll need a way to take care of the other property the hash function has to have which is to map the keywords to the range 0 to $b-1$.

Modulus Operator

The **modulus operator (%)** will be the tool used to change whatever values are calculated for strings into the range 0 to $b-1$. The modulus operator takes a number, divides it by the modulus and returns the remainder.

`<'number'> % <'modulus'> â†' <'remainder'>`

It's like clock arithmetic. The clock has *<modulus>* numbers on it. If you start counting from the top of a clock, and you count around *<number>* of steps, you arrive at the *<remainder>*. For example, if you want to work out **`14 % 12`**, you can think about a clock with 12 numbers. If you count 14 steps, you end up at 2, that is, **`14 % 12 â†' 2`**.



That's the same as the remainder when you divide 14 by 12. When you divide 14 by 12, you get a remainder of 2 since :

$$14 = 12 * 1 + 2$$

The remainder given by modulus is a number between 0 and $\langle \text{modulus} \rangle - 1$. This is good news when we're looking for something for our hash function to give us values between 0 and b-1, where b is the number of buckets.

In terms of the code:

```
print 14 % 12
2
```

Quiz: Modulus Quiz

What is the value of each expression? Try to think of what the answer might be. You can also test it out in the Python interpreter.

- a. $12 \% 3$
- a. `ord('a') % ord('a')` Try to do this without working out w
- a. $(\text{ord}('z') + 3) \% \text{ord}('z')$

[Answer](#)

Quiz: Equivalent Expressions

Which of these expressions are always equivalent to x, where x is any integer between 0 and 10:

1. $x \% 7$

2. `x % 23`
3. `ord(chr(x))`
4. `chr(ord(x))`

[Answer](#)

Bad Hash

The **hash function** takes two inputs, the keyword and the number of buckets and outputs a number between zero and buckets minus one ($b-1$), which gives you the position where that string belongs.

You have seen that the function **ord** takes a one-letter string and maps that to a number.

And you have seen the **modulus** operator, which takes a number and a modulus and outputs the remainder that you get when you divide the number by the modulus.

You can use all of these to define a hash function. Here is an example of a bad hash function:

```
def bad_hash_string (keyword, buckets):  
    return ord(keyword[0]) % buckets # output is the bucket ba
```

Quiz: Bad Hash

Why is **bad_hash_string** a bad hash function?

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

[Answer](#)

Better Hash Function

Now you know that looking at just the first letter does not work very well; it does not use enough buckets, nor does it distribute the keys well. Here is how you can make a better hash function:

Begin with the same properties you had before, a function with two inputs, the keyword and the number of buckets. The output is the hash value in the range of zero to number of buckets minus 1.

The goal is for the keywords to be well distributed across the buckets and every time you hash the same keyword, you will get the same bucket to know quickly where to find it.



In order to make a more robust hash function, you are going to want to look at more than just one letter of the keyword. You want to look at all the letters of the keyword and based on all the letters you want to decide their bucket.

Recall that when you have a list of items you can use the for loop to go through the elements in the list:

```
p = ['a', 'b', ...]
for e in p:
    <'block'>
```

This is also the same for strings:

```
s = "abcd"
for c in s:
    <'block'>
```

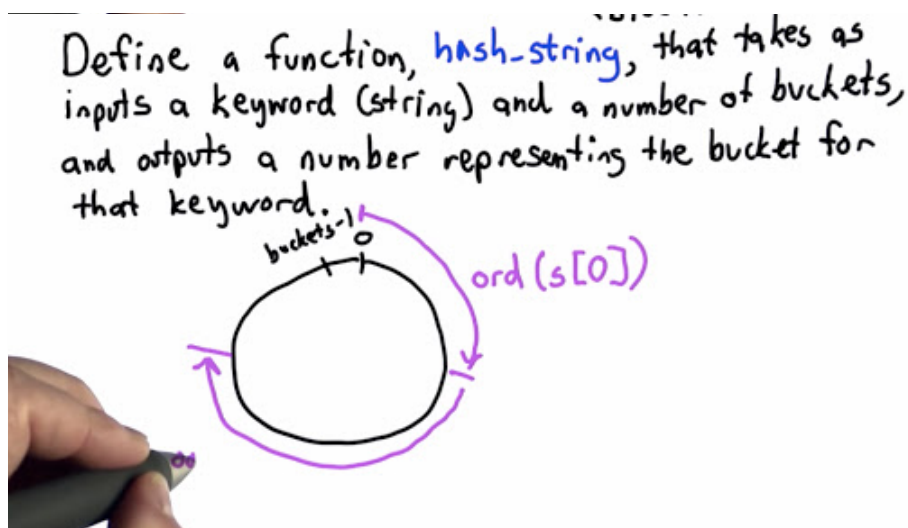
This gives you a way to go through all of the elements in a string. Also recall how you turned single letter strings into numbers with an modulo arithmetic, then you know enough to define a much better hash function.

Quiz: Better Hash Functions

Define a function, **hash_string**, that takes as its input a keyword (string) and a number of buckets, and outputs a number representing the bucket for that keyword. Do this in such a way that it depends on all the characters in the string, not just the first character. Since there are many ways to do this, here are some specifications:

Make the output of **hash_string** a function of all the characters.

You can think about this in terms of modulo arithmetic:



The image shows a circle, which is the size of the number of buckets, and goes from zero to the number of buckets minus 1. For each character start at zero and go around the order of that character, `ord(s[o])`, distance around the circle. Then, keep going so that for each character you go some distance around the circle. The circle can be any size depending on the number of buckets, for example 27.

Which bucket will **a** and **b** land in?

```
hash_string('a', 12) = 1 # 11 will be the last bucket
    ord('a') = 97 # go around the circle eight times, b/c
hash_string('b', 12) = 2
    ord('b') = 98
```

The size of the hash table matters as well as the string:

```
hash_string('a', 13) = 6 # this is because 97 = 13 * 7 + 6
```

What about multi-letter strings?

```
hash_string('au', 12') => 10
    ord('a') = 97
    ord('u') = 117 # add to 97 and modulo the sum, 214 to the
hash_string('udacity', 12) => 11
```

Now try the quiz!

[Answer](#)

Testing Hash Functions

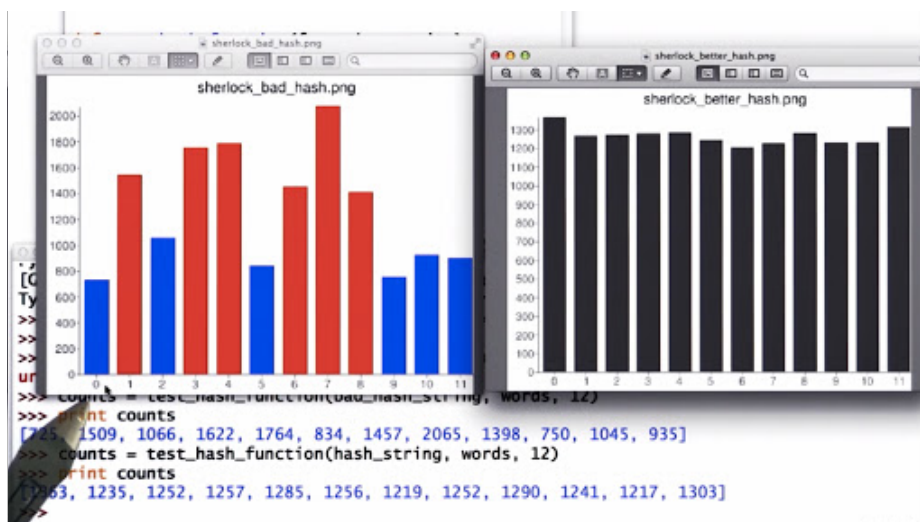
Test your new hash function to make sure it does better than the bad hash function, using the same url from the example before.

```
def test_hash_function(func, keys, size):
    results = [0] * size #this makes a list where all elements are 0
    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

words = get_page('http://www.gutenberg.org/cache/epub/1661/pg1661.html')
counts = test_hash_function(bad_hash_string, words, 12) # obtain counts
print counts
[725, 1509, 1066, 1622, 1764, 834, 1457, 2065, 1398, 750, 1045, 935]
counts = test_hash_function(hash_string, words, 12) # find the counts
print counts
[1363, 1235, 1252, 1257, 1285, 1256, 1219, 1252, 1290, 1241, 1217, 1303]
```

Have a look at the distribution of the keywords into the buckets. Compare the first function, **bad_hash_string**, to the new function, **hash_string**:

NOTE: The hash function is not perfect. Depending on the keys and the number of buckets, it is still possible for all or most of the keywords to end up in a few buckets. It's still being studied....



Now try changing the number of buckets:

```
>>> 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]
>>>
```

Building a good hash function is a very difficult problem. As your tables get larger it is very important to have an efficient hash function and while yours is not the most efficient, it is going to work for your purposes. Check out the website for more examples of how to build an even better hash function.

Quiz: Keywords and Buckets

Assuming our hash function distributes keys perfectly evenly across the buckets, which of the following leaves the time to lookup a keyword essentially unchanged?

There may be more than one answer.

1. double the number of keywords, same # of buckets
2. same number of keywords, double # of buckets
3. double number of keywords, double # of buckets
4. halve number of keywords, same # of buckets
5. halve number of keywords, halve # of buckets

[Answer](#)

Implementing Hash Tables

By now you should understand that the goal of a hash table is to map a keyword and a number of buckets using a **hash_string** function, to a particular bucket, which will contain all of the keywords that map to that location.

Now, try and write the code to do this! You can start with the index you wrote for the previous unit and try to figure out how to implement that with a hash table.

How is this going to change your data structure? Recall the data structure from the last class.

Quiz: Implementing Hash Tables

What data structure should we use to implement our hash table index?

1. [`<word>`, [`<url>`, ...] (/wiki/%3Cword%3E%2C%20%5B%3Curl%3E%2C%20%E2%80%A6), ...]
2. [`<word>`, [[`<url>`, ...], [`<url>`, ...] (/wiki/%3Cword%3E%2C%20%5B%5B%3Curl%3E%2C%20%E2%80%A6%5D%2C%20%5B%3Curl%3E%2C%20%E2%80%A6), ...]
3. [%5B%3Cword%3E%2C%20%5B%3Curl%3E%2C%20%E2%80%A6\), \[`<word>`, \[`<url>`, ..., ...\], ...\]](#)
4. [[`<word>`, `<word>`, ...], [`<url>`, ...] (/wiki/%5B%3Cword%3E%2C%20%3Cword%3E%2C%20%E2%80%A6%20%5D%2C%20%5B%3Curl%3E%2C%20%E2%80%A6), ...]

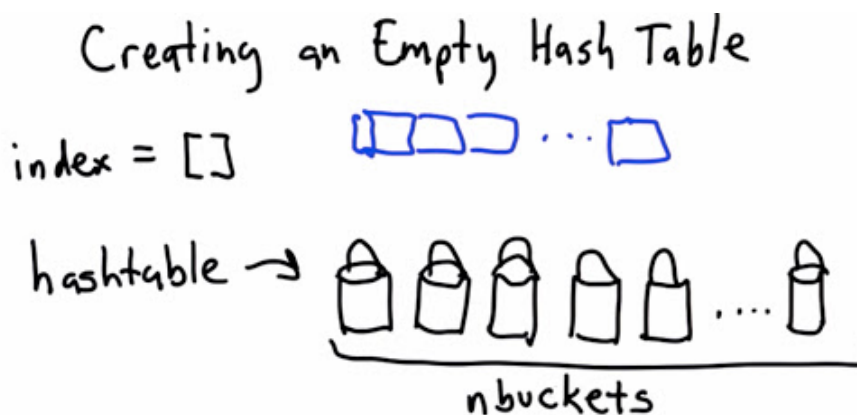
[Answer](#)

Empty Hash Table

The first thing you want to do to implement your hash table index, is figure out how to create an empty hash table. With a simple index, this was really easy, just make an empty list and as you add elements to the list, just add them to the empty list.

```
index = []
```

Unfortunately, this will not work for an empty hash table because you need to start with all the buckets. The initial value for the hash table needs to be a set of empty buckets so that you can do lookups right away. You also want to be able to add elements to your hash table.



If you just started with an empty list, then the first time you looked up a keyword it would say that keyword belongs in bucket 27, but since you don't have a bucket for that you would have to figure out

how to create that bucket. It makes more sense to start by making an empty hash table be a list of buckets, where initially all the buckets are empty, ready and waiting for keywords to be dropped in them. So, what you need is code to create the empty hash table.

Quiz: Empty Hash Table

Define a procedure, **make_hashtable**, that takes as input a number, **nbuckets**, and outputs an empty hash table with **nbuckets** empty buckets.

[Answer](#)

Quiz: The Hard Way

Why does `[[]] * nbuckets` not work to produce our empty hash table?

1. Because it is too easy and we like doing things the hard way.
2. Because each element in the output refers to the same empty list.
3. Because `*` for lists means something different than it does for strings.

[Answer](#)

NOTE: Because all the elements refer to the same empty list. So when you append, you're changing the one object that all the elements refer to. It is possible to change what the list element is referring to by making the element equal to a new list. Then it will not refer to the same empty list.

Finding Buckets

Both operations you will perform on a hash table, i.e. lookup (read) and add (write), depending on first being able to find the right bucket:

- **lookup:** In order to find desired value, you need to know, which bucket to look into.
- **add:** In order to add a value (or overwrite existing one associated with the key, already present in hash table), you need to know which bucket to add your value to (or where to look for the value being overwritten).

Quiz: Finding Buckets

Define a procedure, **hashtable_get_bucket**, that takes two inputs - a hash table and a keyword - and outputs the bucket where the keyword could occur. Note that the bucket returned by the procedure may not contain the searched for keyword in case the keyword is not present in the hash table:

```
hashtable_get_bucket(hashtable, keyword) # => list: bucket
```

Function **hash_string**, defined earlier, may prove useful:

```
hash_string(keyword, nbuckets) # => number: index of bucket
```

Note that there is a little mismatch between those two functions. The function **hashtable_get_bucket** takes hash table and the searched for keyword as its arguments, while the function **hash_string** takes the keyword and **nbuckets**, which is a number (size of the hash table).

The former function does not have any input like this, which means that you will need to determine the size of hash table yourself.

Hint: whole implementation of **hashtable_get_bucket** function might be done on a single line.

[Answer](#)

Quiz: Adding Keywords

Define a procedure:

```
hashtable_add(hhtable, key, value)
```

that adds the key to the hash table (in the correct bucket). Create a new record in the hash table even if the given key already exists. Also note that records in the hash table are lists composed of two elements, where the first one is the key and the second one is the value associated with that key (i.e. **key-value pairs**):

```
[key, value]
```

[Answer](#)

Quiz: Lookup

Define a procedure:

```
hashtable_lookup(hhtable, key)
```

that takes two inputs, a hash table and a key (string), and outputs the value associated with that key. If the key is not in the table, output None.

[Answer](#)

Quiz: Update

Define a procedure:

```
hashtable_update(htable, key, vaue)
```

that updates the value associated with key. If key is already in the table, change the value to the new value. Otherwise add a new entry for the key and value.

[Answer](#)

Operations on Hash Table

You will perform both operations on a hash table, i.e. lookup (read) and add (write), depend on first being able to find the right bucket:

- **lookup:** In order to find desired value, you need to know, which bucket to look into.
- **add:** In order to add a value (or overwrite existing one associated with the key, already present in hash table), you need to know which bucket to add your value to (or where to look for the value being overwritten).

Dictionaries

Now that you've built a hash table for yourself, you'll see an easier way to do it using the built-in Python type **dictionary**. It's an implementation of a hash table, and it's easier to use than defining your own hash table.

You've already seen two complex types in Python, **string** type and **list** type. The dictionary type is another. These three types have many things in common but other things are different.

To create a string, you have a sequence of characters inside quotes ' '. To create a list, you use square brackets [] and have a sequence of elements inside the square brackets, separated by commas. The elements can be of any type, unlike with a string which has to be made up of characters. The dictionary type is created using curly

brackets `{ }`, and consists of **key:value** pairs. The keys can be any immutable type, such as a string or a number, and the values can be of any type.

Recall a string is immutable which means that once it is created, the characters can not be changed. A list is mutable, which means it can be changed once it's been created. A dictionary is also mutable and its **key:value** pairs are updated like in your update function for hash tables. In other words, if the key isn't in the list, it's created, and if it is, it's changed to the new value. The differences, and similarities between a string, list and dictionary are summarised below.

String	List	Dictionary
'hello'	['alpha', 23]	{'hydrogen': 1, 'helium': 2}
sequence of characters	list of elements	set of {key: value} pairs
immutable	mutable	mutable
<code>s[i]</code> = i^{th} character in <code>s</code>	<code>p[i]</code> = i^{th} element of <code>p</code>	<code>d[k]</code> = value associated with key <code>k</code> in <code>d</code>
 <code>s[i] = 'x'</code> 	<code>p[i] = u</code>	<code>d[k] = v</code>
no modification allowed: string is immutable	replace i^{th} element with value <code>u</code>	update value corresponding to key <code>k</code> to become <code>v</code>

Using Dictionaries

You can create a dictionary using `{ }`. For the example below, the key:value pairs are the chemical elements along with their atomic numbers.

```
elements = { 'hydrogen': 1, 'helium': 2, 'carbon': 6 }

print elements
{'helium': 2, 'hydrogen': 1, 'carbon': 6}
```

Unlike with a list, when the elements are printed out, the **key:value** pairs can be in a different order from the order they were entered into the dictionary. When you made a hash table and put elements into it, you saw that the position of the elements in the table was not

necessarily the same order as they were entered as it depended on the key and the hash function. You see the same thing with the dictionary as with your hash table as the dictionary is implemented like a hash table.

When you look up a chemical element in the dictionary, the value associated with that element is returned.

```
print elements['hydrogen']
1
print elements['carbon']
6
```

What do you think will happen when you try to lookup an element which is not in the list?

```
print elements['lithium']
Traceback (most recent call last):
  File "C:\Users\Sarah\Documents\courses\python\dummy.py", line 1, in
    print elements['lithium']
KeyError: 'lithium'
```

You get a [*KeyError*](#) which tells you that "*lithium*" is not in the dictionary. Unlike in your lookup where you defined it to return **None** if the key is not there, you get an error if a key is not in a dictionary and you try to do a lookup on it.

To prevent this error, you can check if a **key** is in the dictionary using `in`, just like with lists. Just like for lists, it will return **True** if the key is in the list and **False** if it is not.

```
print 'lithium' in elements
False
```

As a dictionary is mutable, it can be added to and changed. Using **elements['lithium']** on the left hand side of an assignment does not cause an error even though "**lithium**" is not in the dictionary. Instead it adds the **key:value** pair "**lithium**":**3** to the dictionary.

```
elements['lithium'] = 3
elements['nitrogen'] = 8

print elements
{'helium': 2, 'lithium': 3, 'hydrogen': 1, 'nitrogen': 8, 'carbon': 6}

print element['nitrogen']
8
```

Although this gives the output expected, the atomic number of

"**nitrogen**" is actually 7 and not 8, so it needs to be changed. As "**nitrogen**" is already in the dictionary, this time:

```
elements['nitrogen'] = 7
```

doesn't create a new **key:value** pair, but instead it updates the value associated with "**nitrogen**". To see that, you can see print the value:

```
print element['nitrogen']  
7
```

and the complete dictionary.

```
print elements  
{ 'helium': 2, 'lithium': 3, 'hydrogen': 1, 'nitrogen': 7, 'ca
```

Quiz: Population

Define a dictionary, **population**, that provides information on the world's largest cities. The key is the name of a city (a string), and the associated value is its population in millions.

Shanghai 17.8 Istanbul 13.3 Karachi 13.0 Mumbai 12.5

If you don't happen to live in one of those cities, you might also like to add your hometown and its population or any other cities you might be interested in. If you define your dictionary correctly, you should be able to test it using **print population['Mumbai']** for which you should get the output *12.5*.

[Answer](#)

A Noble Gas

Now to return to the elements dictionary, but this time, to make it more interesting. The chemical element dictionary from earlier just contains elements and their atomic numbers.

```
elements = { 'hydrogen': 1, 'helium': 2, 'carbon': 6 }  
  
elements['lithium'] = 3  
elements['nitrogen'] = 8  
  
print elements['nitrogen']  
elements['nitrogen'] = 7  
print elements['nitrogen']
```

Values don't have to be numbers or strings. They can be anything you want. They can even be other dictionaries. The next example will have atomic symbols as keys with associated values which are dictionaries. First, an empty dictionary is created and then hydrogen, with key **"H"** and helium, with key **"He"** are added to it.

```
elements = {}
elements['H'] = {'name': 'Hydrogen', 'number': 1, 'weight': 1}
elements['He'] = {'name': 'Helium', 'number': 2, 'weight': 4,
                  'noble_gas': True}
```

The code:

```
elements['H'] = {'name': 'Hydrogen', 'number': 1, 'weight': 1}
```

sets **"H"** as the key with associated value the dictionary **{'name': 'Hydrogen', 'number': 1, 'weight': 1.00794}**. This dictionary has three entries with the keys **"name"**, **"number"** and **"weight"**. For helium, **"He"** is the key and a dictionary containing the same keys as **"H"** but with an extra entry which has key **"noble_gas"** and value **True**.

The dictionary of **key:value** pairs, **atomic_symbol: {dictionary}** is shown below.

```
print elements
{'H': {'name': 'Hydrogen', 'weight': 1.00794, 'number': 1},
 'He': {'noble_gas': True, 'name': 'Helium', 'weight': 4.00260}}
```

To see the element hydrogen, look up its key which is 'H'.

```
print elements['H']
{'name': 'Hydrogen', 'weight': 1.00794, 'number': 1}
```

Note that the elements appear in a different order from the order they were input as **elements['H']** is a dictionary.

To look up the name of the element with symbol H, use:

```
print elements['H']['name']
Hydrogen
```

where **elements['H']** is itself a dictionary and **"name"** is one of its keys.

You could also lookup the weights of hydrogen and of helium, or check if helium is a noble gas.

```
print elements['H']['weight']
```

```
1.00794
print elements['He']['weight']
4.002602
print elements['He']['noble gas']
True
```

What happens if you try to check if hydrogen is a noble gas?

```
print elements['H']['noble gas']
Traceback (most recent call last):
  File "C:\Users\Sarah\Documents\courses\python\dummy.py", line 1, in
    print elements['H']['noble gas']
KeyError: 'noble gas'
```

It's the same error as for the attempted lookup of lithium in the dictionary of elements which didn't include it. It tries to look up **"noble gas"** but it doesn't exist in the dictionary which is associated with the key **"H"**.

Modifying the Search Engine

Modifying the search engine code from the previous unit to use dictionary indexes instead of list indexes has the advantage of doing lookups in constant time rather than linear time.

Quiz: Modifying the Search Engine

Which of the procedures in your search engine do you need to change to make use of a dictionary index instead of a list index?

1. `get_all_links`
2. `crawl_web`
3. `add_page_to_index`
4. `add_to_index`
5. `lookup`

Code

```
def get_all_links(page):
    links = []
    while True:
        url, endpos = get_next_target(page)
        if url:
            links.append(url)
            page = page[endpos:]
        else:
            break
    return links
```

```

def crawl_web(seed):
    tocrawl = [seed]
    crawled = []
    index = []
    while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:
            content = get_page(page)
            add_page_to_index(index, page, content)
            union(tocrawl, get_all_links(content))
            crawled.append(page)
    return index

def add_page_to_index(index, url, content):
    words = content.split()
    for word in words:
        add_to_index(index, word, url)

def add_to_index(index, keyword, url):
    for entry in index:
        if entry[0] == keyword:
            entry[1].append(url)
            return
    # not found, add new keyword to index
    index.append([keyword, [url]])

def lookup(index, keyword):
    for entry in index:
        if entry[0] == keyword:
            return entry[1]
    return None

```

[Answer](#)

Quiz: Changing Lookup

Change the lookup procedure to use a dictionary rather than a list index. This does not require any loop. Make sure an error is not raised if the keyword is not in the index; in that case, return None.

[Answer](#)

Changing Lookup

Congratulations! You have completed unit 5. You now have a search engine that can respond to queries quickly, no matter how large the index gets. You did so by replacing the list data structure with a

hash table, which can respond to a query in a time that does not increase even if the index increases.

The problem of measuring cost, analyzing algorithms and designing algorithms that work well when the input size scales, is one of the most important and interesting problems in computer science.

Now, all you have left to do for your search engine is to figure out a way to find the best page for a given query instead of just finding all of the pages that have that keyword. This is what you will learn in unit 6.
